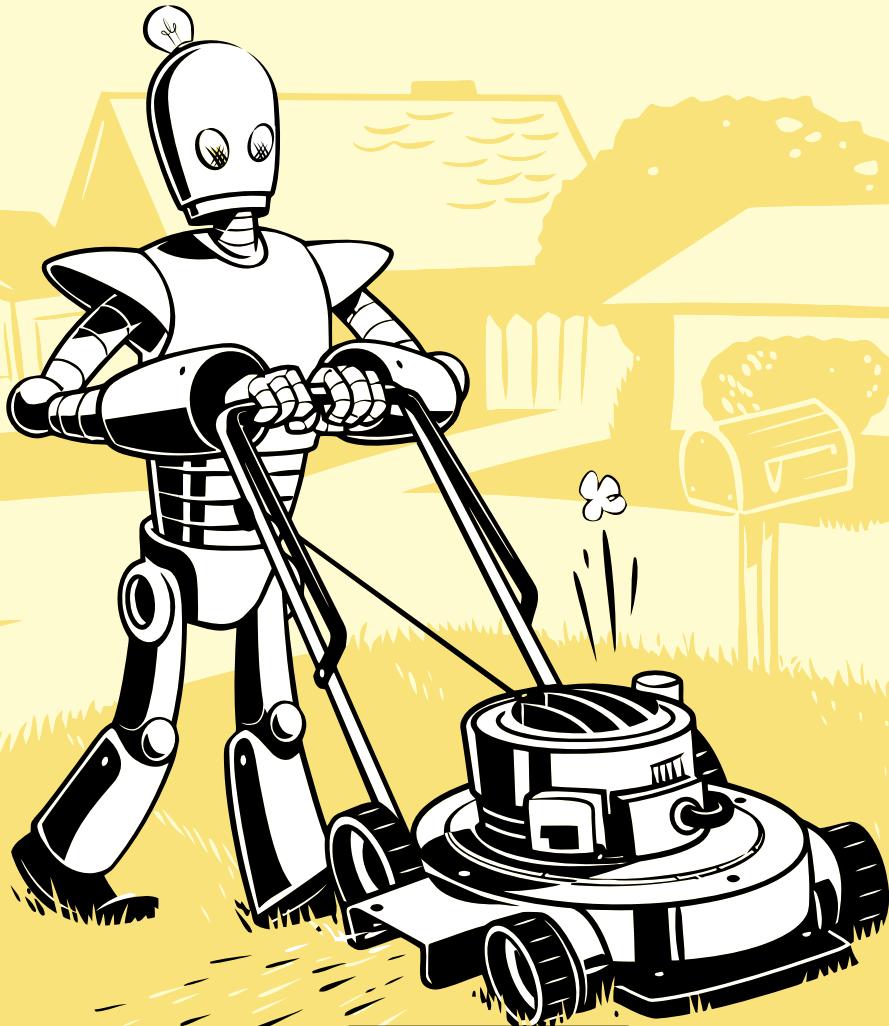


AUTOMATE THE BORING STUFF WITH PYTHON

PRACTICAL PROGRAMMING
FOR TOTAL BEGINNERS

AL SWEIGART



**AUTOMATE THE BORING STUFF
WITH PYTHON**

AUTOMATE THE BORING STUFF WITH PYTHON

**Practical Programming
for Total Beginners**

by Al Sweigart



San Francisco

AUTOMATE THE BORING STUFF WITH PYTHON. Copyright © 2015 by Al Sweigart.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Printed in USA

Second printing

19 18 17 16 15 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-599-4

ISBN-13: 978-1-59327-599-0



Publisher: William Pollock

Production Editor: Laurel Chun

Cover Illustration: Josh Ellingson

Interior Design: Octopod Studios

Developmental Editors: Jennifer Griffith-Delgado, Greg Poulos, and Leslie Shen

Technical Reviewer: Ari Lacenski

Copyeditor: Kim Wimpsett

Compositor: Susan Glinert Stevens

Proofreader: Lisa Devoto Farrell

Indexer: BIM Indexing and Proofreading Services

For information on distribution, translations, or bulk sales,
please contact No Starch Press, Inc. directly:

No Starch Press, Inc.

245 8th Street, San Francisco, CA 94103

phone: 415.863.9900; info@nostarch.com

www.nostarch.com

Library of Congress Control Number: 2014953114

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

For my nephew Jack

About the Author

Al Sweigart is a software developer and tech book author living in San Francisco. Python is his favorite programming language, and he is the developer of several open source modules for it. His other books are freely available under a Creative Commons license on his website <http://www.inventwithpython.com/>. His cat weighs 14 pounds.

About the Tech Reviewer

Ari Lacenski is a developer of Android applications and Python software. She lives in San Francisco, where she writes about Android programming at <http://gradlewhy.ghost.io/> and mentors with Women Who Code. She's also a folk guitarist.

BRIEF CONTENTS

Acknowledgments	xxiii
Introduction	1
PART I: PYTHON PROGRAMMING BASICS.....	11
Chapter 1: Python Basics	13
Chapter 2: Flow Control	31
Chapter 3: Functions	61
Chapter 4: Lists	79
Chapter 5: Dictionaries and Structuring Data.....	105
Chapter 6: Manipulating Strings	123
PART II: AUTOMATING TASKS.....	145
Chapter 7: Pattern Matching with Regular Expressions	147
Chapter 8: Reading and Writing Files	173
Chapter 9: Organizing Files	197
Chapter 10: Debugging	215
Chapter 11: Web Scraping	233
Chapter 12: Working with Excel Spreadsheets	265
Chapter 13: Working with PDF and Word Documents	295
Chapter 14: Working with CSV Files and JSON Data	319
Chapter 15: Keeping Time, Scheduling Tasks, and Launching Programs.....	335
Chapter 16: Sending Email and Text Messages.....	361
Chapter 17: Manipulating Images	387
Chapter 18: Controlling the Keyboard and Mouse with GUI Automation	413

Appendix A: Installing Third-Party Modules	441
Appendix B: Running Programs	443
Appendix C: Answers to the Practice Questions	447
Index	461

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

xxiii

INTRODUCTION

Whom Is This Book For?	2
Conventions	2
What Is Programming?	3
What Is Python?	4
Programmers Don't Need to Know Much Math	4
Programming Is a Creative Activity	5
About This Book	5
Downloading and Installing Python	6
Starting IDLE	7
The Interactive Shell	8
How to Find Help	8
Asking Smart Programming Questions	9
Summary	10

PART I: PYTHON PROGRAMMING BASICS

11

1 PYTHON BASICS

Entering Expressions into the Interactive Shell	14
The Integer, Floating-Point, and String Data Types	16
String Concatenation and Replication	17
Storing Values in Variables	18
Assignment Statements	18
Variable Names	20
Your First Program	21
Dissecting Your Program	22
Comments	23
The print() Function	23
The input() Function	23
Printing the User's Name	24
The len() Function	24
The str(), int(), and float() Functions	25
Summary	28
Practice Questions	28

2 FLOW CONTROL

Boolean Values	32
Comparison Operators	33
Boolean Operators	35

Binary Boolean Operators	35
The not Operator	36
Mixing Boolean and Comparison Operators	36
Elements of Flow Control	37
Conditions	37
Blocks of Code	37
Program Execution	38
Flow Control Statements	38
if Statements	38
else Statements	39
elif Statements	40
while Loop Statements	45
break Statements	49
continue Statements	50
for Loops and the range() Function	53
Importing Modules	57
from import Statements	58
Ending a Program Early with sys.exit()	58
Summary	58
Practice Questions	59

3 FUNCTIONS 61

def Statements with Parameters	63
Return Values and return Statements	63
The None Value	65
Keyword Arguments and print()	65
Local and Global Scope	67
Local Variables Cannot Be Used in the Global Scope	67
Local Scopes Cannot Use Variables in Other Local Scopes	68
Global Variables Can Be Read from a Local Scope	69
Local and Global Variables with the Same Name	69
The global Statement	70
Exception Handling	72
A Short Program: Guess the Number	74
Summary	76
Practice Questions	76
Practice Projects	77
The Collatz Sequence	77
Input Validation	77

4 LISTS 79

The List Data Type	80
Getting Individual Values in a List with Indexes	80
Negative Indexes	82
Getting Sublists with Slices	82
Getting a List's Length with len()	83
Changing Values in a List with Indexes	83

List Concatenation and List Replication	83
Removing Values from Lists with del Statements	84
Working with Lists	84
Using for Loops with Lists	86
The in and not in Operators	87
The Multiple Assignment Trick	87
Augmented Assignment Operators	88
Methods	89
Finding a Value in a List with the index() Method	89
Adding Values to Lists with the append() and insert() Methods	89
Removing Values from Lists with remove()	90
Sorting the Values in a List with the sort() Method	91
Example Program: Magic 8 Ball with a List	92
List-like Types: Strings and Tuples	93
Mutable and Immutable Data Types	94
The Tuple Data Type	96
Converting Types with the list() and tuple() Functions	97
References	97
Passing References	100
The copy Module's copy() and deepcopy() Functions	100
Summary	101
Practice Questions	102
Practice Projects	102
Comma Code	102
Character Picture Grid	103

5	DICTIONARIES AND STRUCTURING DATA	105
The Dictionary Data Type	105	
Dictionaries vs. Lists	106	
The keys(), values(), and items() Methods	107	
Checking Whether a Key or Value Exists in a Dictionary	109	
The get() Method	109	
The setdefault() Method	110	
Pretty Printing	111	
Using Data Structures to Model Real-World Things	112	
A Tic-Tac-Toe Board	113	
Nested Dictionaries and Lists	117	
Summary	119	
Practice Questions	119	
Practice Projects	120	
Fantasy Game Inventory	120	
List to Dictionary Function for Fantasy Game Inventory	120	

6	MANIPULATING STRINGS	123
Working with Strings	123	
String Literals	124	
Indexing and Slicing Strings	126	
The in and not in Operators with Strings	127	

Useful String Methods	127
The upper(), lower(), isupper(), and islower() String Methods	128
The isX String Methods	129
The startswith() and endswith() String Methods	131
The join() and split() String Methods	131
Justifying Text with rjust(), ljust(), and center()	133
Removing Whitespace with strip(), rstrip(), and lstrip()	134
Copying and Pasting Strings with the pyperclip Module.	135
Project: Password Locker	136
Step 1: Program Design and Data Structures	136
Step 2: Handle Command Line Arguments	137
Step 3: Copy the Right Password	137
Project: Adding Bullets to Wiki Markup	139
Step 1: Copy and Paste from the Clipboard	139
Step 2: Separate the Lines of Text and Add the Star	140
Step 3: Join the Modified Lines	141
Summary	141
Practice Questions	142
Practice Project	142
Table Printer	142

PART II: AUTOMATING TASKS 145

7	
PATTERN MATCHING WITH REGULAR EXPRESSIONS	147
Finding Patterns of Text Without Regular Expressions	148
Finding Patterns of Text with Regular Expressions	150
Creating Regex Objects	150
Matching Regex Objects	151
Review of Regular Expression Matching	152
More Pattern Matching with Regular Expressions	152
Grouping with Parentheses.	152
Matching Multiple Groups with the Pipe.	153
Optional Matching with the Question Mark	154
Matching Zero or More with the Star.	155
Matching One or More with the Plus	155
Matching Specific Repetitions with Curly Brackets	156
Greedy and Nongreedy Matching	156
The.findall() Method	157
Character Classes	158
Making Your Own Character Classes	159
The Caret and Dollar Sign Characters	159
The Wildcard Character	160
Matching Everything with Dot-Star.	161
Matching Newlines with the Dot Character	162
Review of Regex Symbols	162
Case-Insensitive Matching	163

Substituting Strings with the <code>sub()</code> Method	163
Managing Complex Regexes	164
Combining <code>re.IGNORECASE</code> , <code>re.DOTALL</code> , and <code>re.VERBOSE</code>	164
Project: Phone Number and Email Address Extractor	165
Step 1: Create a Regex for Phone Numbers	166
Step 2: Create a Regex for Email Addresses	166
Step 3: Find All Matches in the Clipboard Text	167
Step 4: Join the Matches into a String for the Clipboard	168
Running the Program	169
Ideas for Similar Programs	169
Summary	169
Practice Questions	170
Practice Projects	171
Strong Password Detection	171
Regex Version of <code>strip()</code>	171

8 READING AND WRITING FILES 173

Files and File Paths	173
Backslash on Windows and Forward Slash on OS X and Linux	174
The Current Working Directory	175
Absolute vs. Relative Paths	175
Creating New Folders with <code>os.makedirs()</code>	176
The <code>os.path</code> Module	177
Handling Absolute and Relative Paths	177
Finding File Sizes and Folder Contents	179
Checking Path Validity	180
The File Reading/Writing Process	180
Opening Files with the <code>open()</code> Function	181
Reading the Contents of Files	182
Writing to Files	183
Saving Variables with the <code>shelve</code> Module	184
Saving Variables with the <code>pprint.pformat()</code> Function	185
Project: Generating Random Quiz Files	186
Step 1: Store the Quiz Data in a Dictionary	187
Step 2: Create the Quiz File and Shuffle the Question Order	188
Step 3: Create the Answer Options	189
Step 4: Write Content to the Quiz and Answer Key Files	189
Project: Multiclipboard	191
Step 1: Comments and Shelf Setup	192
Step 2: Save Clipboard Content with a Keyword	192
Step 3: List Keywords and Load a Keyword's Content	193
Summary	194
Practice Questions	194
Practice Projects	194
Extending the Multiclipboard	194
Mad Libs	195
Regex Search	195

ORGANIZING FILES

The shutil Module	198
Copying Files and Folders	198
Moving and Renaming Files and Folders	199
Permanently Deleting Files and Folders	200
Safe Deletes with the send2trash Module	201
Walking a Directory Tree	202
Compressing Files with the zipfile Module	203
Reading ZIP Files	204
Extracting from ZIP Files	205
Creating and Adding to ZIP Files	205
Project: Renaming Files with American-Style Dates to European-Style Dates	206
Step 1: Create a Regex for American-Style Dates	206
Step 2: Identify the Date Parts from the Filenames	207
Step 3: Form the New Filename and Rename the Files	209
Ideas for Similar Programs	209
Project: Backing Up a Folder into a ZIP File	209
Step 1: Figure Out the ZIP File's Name	210
Step 2: Create the New ZIP File	211
Step 3: Walk the Directory Tree and Add to the ZIP File	211
Ideas for Similar Programs	212
Summary	212
Practice Questions	213
Practice Projects	213
Selective Copy	213
Deleting Unneeded Files	213
Filling in the Gaps	214

DEBUGGING

Raising Exceptions	216
Getting the Traceback as a String	217
Assertions	219
Using an Assertion in a Traffic Light Simulation	219
Disabling Assertions	221
Logging	221
Using the logging Module	221
Don't Debug with print()	223
Logging Levels	223
Disabling Logging	224
Logging to a File	225
IDLE's Debugger	225
Go	226
Step	226
Over	226
Out	227

Quit	227
Debugging a Number Adding Program	227
Breakpoints	229
Summary	231
Practice Questions	231
Practice Project	232
Debugging Coin Toss	232

11 WEB SCRAPING 233

Project: <code>maplt.py</code> with the <code>webbrowser</code> Module	234
Step 1: Figure Out the URL	234
Step 2: Handle the Command Line Arguments	235
Step 3: Handle the Clipboard Content and Launch the Browser	236
Ideas for Similar Programs	236
Downloading Files from the Web with the <code>requests</code> Module	237
Downloading a Web Page with the <code>requests.get()</code> Function	237
Checking for Errors	238
Saving Downloaded Files to the Hard Drive	239
HTML	240
Resources for Learning HTML	240
A Quick Refresher	240
Viewing the Source HTML of a Web Page	241
Opening Your Browser's Developer Tools	242
Using the Developer Tools to Find HTML Elements	244
Parsing HTML with the <code>BeautifulSoup</code> Module	245
Creating a <code>BeautifulSoup</code> Object from HTML	245
Finding an Element with the <code>select()</code> Method	246
Getting Data from an Element's Attributes	248
Project: "I'm Feeling Lucky" Google Search	248
Step 1: Get the Command Line Arguments and Request the Search Page	249
Step 2: Find All the Results	249
Step 3: Open Web Browsers for Each Result	250
Ideas for Similar Programs	251
Project: Downloading All XKCD Comics	251
Step 1: Design the Program	252
Step 2: Download the Web Page	253
Step 3: Find and Download the Comic Image	254
Step 4: Save the Image and Find the Previous Comic	255
Ideas for Similar Programs	256
Controlling the Browser with the <code>selenium</code> Module	256
Starting a Selenium-Controlled Browser	256
Finding Elements on the Page	257
Clicking the Page	259
Filling Out and Submitting Forms	259
Sending Special Keys	260
Clicking Browser Buttons	261
More Information on Selenium	261

Summary	261
Practice Questions	261
Practice Projects	262
Command Line Emailer	262
Image Site Downloader	263
2048	263
Link Verification	263

12 WORKING WITH EXCEL SPREADSHEETS 265

Excel Documents	266
Installing the openpyxl Module	266
Reading Excel Documents	266
Opening Excel Documents with OpenPyXL	267
Getting Sheets from the Workbook	268
Getting Cells from the Sheets	268
Converting Between Column Letters and Numbers	270
Getting Rows and Columns from the Sheets	270
Workbooks, Sheets, Cells	272
Project: Reading Data from a Spreadsheet	272
Step 1: Read the Spreadsheet Data	273
Step 2: Populate the Data Structure	274
Step 3: Write the Results to a File	275
Ideas for Similar Programs	276
Writing Excel Documents	277
Creating and Saving Excel Documents	277
Creating and Removing Sheets	278
Writing Values to Cells	278
Project: Updating a Spreadsheet	279
Step 1: Set Up a Data Structure with the Update Information	280
Step 2: Check All Rows and Update Incorrect Prices	281
Ideas for Similar Programs	281
Setting the Font Style of Cells	282
Font Objects	282
Formulas	284
Adjusting Rows and Columns	285
Setting Row Height and Column Width	285
Merging and Unmerging Cells	286
Freeze Panes	287
Charts	288
Summary	290
Practice Questions	291
Practice Projects	291
Multiplication Table Maker	291
Blank Row Inserter	292
Spreadsheet Cell Inverter	292
Text Files to Spreadsheet	293
Spreadsheet to Text Files	293

13	WORKING WITH PDF AND WORD DOCUMENTS	295
PDF Documents	295	
Extracting Text from PDFs	296	
Decrypting PDFs	297	
Creating PDFs.	298	
Project: Combining Select Pages from Many PDFs	303	
Step 1: Find All PDF Files	304	
Step 2: Open Each PDF	304	
Step 3: Add Each Page	305	
Step 4: Save the Results	305	
Ideas for Similar Programs	306	
Word Documents	306	
Reading Word Documents	307	
Getting the Full Text from a .docx File	308	
Styling Paragraph and Run Objects.	309	
Creating Word Documents with Nondefault Styles	310	
Run Attributes	311	
Writing Word Documents	312	
Adding Headings	314	
Adding Line and Page Breaks.	315	
Adding Pictures.	315	
Summary	316	
Practice Questions	316	
Practice Projects.	317	
PDF Paranoia	317	
Custom Invitations as Word Documents	317	
Brute-Force PDF Password Breaker.	318	
14	WORKING WITH CSV FILES AND JSON DATA	319
The csv Module	320	
Reader Objects.	321	
Reading Data from Reader Objects in a for Loop	322	
Writer Objects	322	
The delimiter and lineterminator Keyword Arguments.	323	
Project: Removing the Header from CSV Files	324	
Step 1: Loop Through Each CSV File	325	
Step 2: Read in the CSV File	325	
Step 3: Write Out the CSV File Without the First Row	326	
Ideas for Similar Programs	327	
JSON and APIs	327	
The json Module	328	
Reading JSON with the loads() Function	328	
Writing JSON with the dumps() Function	329	
Project: Fetching Current Weather Data	329	
Step 1: Get Location from the Command Line Argument.	330	
Step 2: Download the JSON Data.	330	
Step 3: Load JSON Data and Print Weather.	331	
Ideas for Similar Programs	332	
Summary	333	

Practice Questions	333
Practice Project	333
Excel-to-CSV Converter	333

15 **KEEPING TIME, SCHEDULING TASKS, AND LAUNCHING PROGRAMS**

	335
The time Module	336
The time.time() Function	336
The time.sleep() Function	337
Rounding Numbers	338
Project: Super Stopwatch	338
Step 1: Set Up the Program to Track Times	339
Step 2: Track and Print Lap Times	339
Ideas for Similar Programs	340
The datetime Module	341
The timedelta Data Type	342
Pausing Until a Specific Date	344
Converting datetime Objects into Strings	344
Converting Strings into datetime Objects	345
Review of Python’s Time Functions	346
Multithreading	347
Passing Arguments to the Thread’s Target Function	348
Concurrency Issues	349
Project: Multithreaded XKCD Downloader	350
Step 1: Modify the Program to Use a Function	350
Step 2: Create and Start Threads	351
Step 3: Wait for All Threads to End	352
Launching Other Programs from Python	352
Passing Command Line Arguments to Popen()	354
Task Scheduler, launchd, and cron	354
Opening Websites with Python	355
Running Other Python Scripts	355
Opening Files with Default Applications	355
Project: Simple Countdown Program	357
Step 1: Count Down	357
Step 2: Play the Sound File	357
Ideas for Similar Programs	358
Summary	358
Practice Questions	359
Practice Projects	359
Prettified Stopwatch	360
Scheduled Web Comic Downloader	360

16 **SENDING EMAIL AND TEXT MESSAGES**

	361
SMTP	362
Sending Email	362
Connecting to an SMTP Server	363
Sending the SMTP “Hello” Message	364

Starting TLS Encryption	364
Logging in to the SMTP Server	364
Sending an Email	365
Disconnecting from the SMTP Server	366
IMAP	366
Retrieving and Deleting Emails with IMAP	366
Connecting to an IMAP Server	367
Logging in to the IMAP Server	368
Searching for Email	368
Fetching an Email and Marking It As Read	372
Getting Email Addresses from a Raw Message	373
Getting the Body from a Raw Message	374
Deleting Emails	375
Disconnecting from the IMAP Server	375
Project: Sending Member Dues Reminder Emails	376
Step 1: Open the Excel File	376
Step 2: Find All Unpaid Members	378
Step 3: Send Customized Email Reminders	378
Sending Text Messages with Twilio	380
Signing Up for a Twilio Account	380
Sending Text Messages	381
Project: "Just Text Me" Module	383
Summary	384
Practice Questions	384
Practice Projects	385
Random Chore Assignment Emailer	385
Umbrella Reminder	385
Auto Unsubscriber	385
Controlling Your Computer Through Email	386

17		MANIPULATING IMAGES	387
Computer Image Fundamentals	388		
Colors and RGBA Values	388		
Coordinates and Box Tuples	389		
Manipulating Images with Pillow	390		
Working with the Image Data Type	392		
Cropping Images	393		
Copying and Pasting Images onto Other Images	394		
Resizing an Image	397		
Rotating and Flipping Images	398		
Changing Individual Pixels	400		
Project: Adding a Logo	401		
Step 1: Open the Logo Image	401		
Step 2: Loop Over All Files and Open Images	402		
Step 3: Resize the Images	403		
Step 4: Add the Logo and Save the Changes	404		
Ideas for Similar Programs	406		
Drawing on Images	406		
Drawing Shapes	406		
Drawing Text	408		

Summary	410
Practice Questions	410
Practice Projects	411
Extending and Fixing the Chapter Project Programs	411
Identifying Photo Folders on the Hard Drive	411
Custom Seating Cards	412

18

CONTROLLING THE KEYBOARD AND MOUSE WITH GUI AUTOMATION

413

Installing the pyautogui Module	414
Staying on Track	414
Shutting Down Everything by Logging Out	414
Pauses and Fail-Safes	415
Controlling Mouse Movement	415
Moving the Mouse	416
Getting the Mouse Position	417
Project: “Where Is the Mouse Right Now?”	417
Step 1: Import the Module	418
Step 2: Set Up the Quit Code and Infinite Loop	418
Step 3: Get and Print the Mouse Coordinates	418
Controlling Mouse Interaction	419
Clicking the Mouse	420
Dragging the Mouse	420
Scrolling the Mouse	422
Working with the Screen	423
Getting a Screenshot	423
Analyzing the Screenshot	424
Project: Extending the mouseNow Program	424
Image Recognition	425
Controlling the Keyboard	426
Sending a String from the Keyboard	426
Key Names	427
Pressing and Releasing the Keyboard	428
Hotkey Combinations	429
Review of the PyAutoGUI Functions	430
Project: Automatic Form Filler	430
Step 1: Figure Out the Steps	432
Step 2: Set Up Coordinates	432
Step 3: Start Typing Data	434
Step 4: Handle Select Lists and Radio Buttons	435
Step 5: Submit the Form and Wait	436
Summary	437
Practice Questions	438
Practice Projects	438
Looking Busy	438
Instant Messenger Bot	438
Game-Playing Bot Tutorial	439

A		
INSTALLING THIRD-PARTY MODULES		441
The pip Tool	441	
Installing Third-Party Modules	442	
B		
RUNNING PROGRAMS		443
Shebang Line.	443	
Running Python Programs on Windows.	444	
Running Python Programs on OS X and Linux	445	
Running Python Programs with Assertions Disabled	445	
C		
ANSWERS TO THE PRACTICE QUESTIONS		447
Chapter 1	448	
Chapter 2	448	
Chapter 3	450	
Chapter 4	450	
Chapter 5	451	
Chapter 6	451	
Chapter 7	452	
Chapter 8	453	
Chapter 9	453	
Chapter 10	454	
Chapter 11	455	
Chapter 12	456	
Chapter 13	456	
Chapter 14	457	
Chapter 15	457	
Chapter 16	458	
Chapter 17	458	
Chapter 18	458	
INDEX		461

ACKNOWLEDGMENTS

I couldn't have written a book like this without the help of a lot of people. I'd like to thank Bill Pollock; my editors, Laurel Chun, Leslie Shen, Greg Poulos, and Jennifer Griffith-Delgado; and the rest of the staff at No Starch Press for their invaluable help. Thanks to my tech reviewer, Ari Lacenski, for great suggestions, edits, and support.

Many thanks to our Benevolent Dictator For Life, Guido van Rossum, and everyone at the Python Software Foundation for their great work. The Python community is the best one I've found in the tech industry.

Finally, I would like to thank my family, friends, and the gang at Shotwell's for not minding the busy life I've had while writing this book. Cheers!

INTRODUCTION



“You’ve just done in two hours what it takes the three of us two days to do.” My college roommate was working at a retail electronics store in the early 2000s. Occasionally, the store would receive a spreadsheet of thousands of product prices from its competitor. A team of three employees would print the spreadsheet onto a thick stack of paper and split it among themselves. For each product price, they would look up their store’s price and note all the products that their competitors sold for less. It usually took a couple of days.

“You know, I could write a program to do that if you have the original file for the printouts,” my roommate told them, when he saw them sitting on the floor with papers scattered and stacked around them.

After a couple of hours, he had a short program that read a competitor’s price from a file, found the product in the store’s database, and noted whether the competitor was cheaper. He was still new to programming, and

he spent most of his time looking up documentation in a programming book. The actual program took only a few seconds to run. My roommate and his co-workers took an extra-long lunch that day.

This is the power of computer programming. A computer is like a Swiss Army knife that you can configure for countless tasks. Many people spend hours clicking and typing to perform repetitive tasks, unaware that the machine they're using could do their job in seconds if they gave it the right instructions.

Whom Is This Book For?

Software is at the core of so many of the tools we use today: Nearly everyone uses social networks to communicate, many people have Internet-connected computers in their phones, and most office jobs involve interacting with a computer to get work done. As a result, the demand for people who can code has skyrocketed. Countless books, interactive web tutorials, and developer boot camps promise to turn ambitious beginners into software engineers with six-figure salaries.

This book is not for those people. It's for everyone else.

On its own, this book won't turn you into a professional software developer any more than a few guitar lessons will turn you into a rock star. But if you're an office worker, administrator, academic, or anyone else who uses a computer for work or fun, you will learn the basics of programming so that you can automate simple tasks such as the following:

- Moving and renaming thousands of files and sorting them into folders
- Filling out online forms, no typing required
- Downloading files or copy text from a website whenever it updates
- Having your computer text you custom notifications
- Updating or formatting Excel spreadsheets
- Checking your email and sending out prewritten responses

These tasks are simple but time-consuming for humans, and they're often so trivial or specific that there's no ready-made software to perform them. Armed with a little bit of programming knowledge, you can have your computer do these tasks for you.

Conventions

This book is not designed as a reference manual; it's a guide for beginners. The coding style sometimes goes against best practices (for example, some programs use global variables), but that's a trade-off to make the code simpler to learn. This book is made for people to write throwaway code, so there's not much time spent on style and elegance. Sophisticated programming concepts—like object-oriented programming, list comprehensions,

and generators—aren’t covered because of the complexity they add. Veteran programmers may point out ways the code in this book could be changed to improve efficiency, but this book is mostly concerned with getting programs to work with the least amount of effort.

What Is Programming?

Television shows and films often show programmers furiously typing cryptic streams of 1s and 0s on glowing screens, but modern programming isn’t that mysterious. *Programming* is simply the act of entering instructions for the computer to perform. These instructions might crunch some numbers, modify text, look up information in files, or communicate with other computers over the Internet.

All programs use basic instructions as building blocks. Here are a few of the most common ones, in English:

“Do this; then do that.”

“If this condition is true, perform this action; otherwise, do that action.”

“Do this action that number of times.”

“Keep doing that until this condition is true.”

You can combine these building blocks to implement more intricate decisions, too. For example, here are the programming instructions, called the *source code*, for a simple program written in the Python programming language. Starting at the top, the Python software runs each line of code (some lines are run only *if* a certain condition is true or *else* Python runs some other line) until it reaches the bottom.

```
❶ passwordFile = open('SecretPasswordFile.txt')
❷ secretPassword = passwordFile.read()
❸ print('Enter your password.')
    typedPassword = input()
❹ if typedPassword == secretPassword:
❺     print('Access granted')
❻     if typedPassword == '12345':
❼         print('That password is one that an idiot puts on their luggage.')
❼ else:
❽     print('Access denied')
```

You might not know anything about programming, but you could probably make a reasonable guess at what the previous code does just by reading it. First, the file *SecretPasswordFile.txt* is opened ❶, and the secret password in it is read ❷. Then, the user is prompted to input a password (from the keyboard) ❸. These two passwords are compared ❹, and if they’re the same, the program prints *Access granted* to the screen ❺. Next, the program checks to see whether the password is *12345* ❻ and hints that this choice might not be the best for a password ❼. If the passwords are not the same, the program prints *Access denied* to the screen ❽.

What Is Python?

Python refers to the Python programming language (with syntax rules for writing what is considered valid Python code) and the Python interpreter software that reads source code (written in the Python language) and performs its instructions. The Python interpreter is free to download from <http://python.org/>, and there are versions for Linux, OS X, and Windows.

The name Python comes from the surreal British comedy group Monty Python, not from the snake. Python programmers are affectionately called Pythonistas, and both Monty Python and serpentine references usually pepper Python tutorials and documentation.

Programmers Don't Need to Know Much Math

The most common anxiety I hear about learning to program is that people think it requires a lot of math. Actually, most programming doesn't require math beyond basic arithmetic. In fact, being good at programming isn't that different from being good at solving Sudoku puzzles.

To solve a Sudoku puzzle, the numbers 1 through 9 must be filled in for each row, each column, and each 3×3 interior square of the full 9×9 board. You find a solution by applying deduction and logic from the starting numbers. For example, since 5 appears in the top left of the Sudoku puzzle shown in Figure 0-1, it cannot appear elsewhere in the top row, in the leftmost column, or in the top-left 3×3 square. Solving one row, column, or square at a time will provide more number clues for the rest of the puzzle.

5	3			7					
6			1	9	5				
	9	8				6			
8				6				3	
4		8		3				1	
7			2				6		
	6				2	8			
		4	1	9				5	
			8		7	9			

5	3	4	6	7	8	9	1	2	
6	7	2	1	9	5	3	4	8	
1	9	8	3	4	2	5	6	7	
8	5	9	7	6	1	4	2	3	
4	2	6	8	5	3	7	9	1	
7	1	3	9	2	4	8	5	6	
9	6	1	5	3	7	2	8	4	
2	8	7	4	1	9	6	3	5	
3	4	5	2	8	6	1	7	9	

Figure 0-1: A new Sudoku puzzle (left) and its solution (right). Despite using numbers, Sudoku doesn't involve much math. (Images © Wikimedia Commons)

Just because Sudoku involves numbers doesn't mean you have to be good at math to figure out the solution. The same is true of programming. Like solving a Sudoku puzzle, writing programs involves breaking down a problem into individual, detailed steps. Similarly, when *debugging* programs (that is, finding and fixing errors), you'll patiently observe what the program is doing and find the cause of the bugs. And like all skills, the more you program, the better you'll become.

Programming Is a Creative Activity

Programming is a creative task, somewhat like constructing a castle out of LEGO bricks. You start with a basic idea of what you want your castle to look like and inventory your available blocks. Then you start building. Once you've finished building your program, you can pretty up your code just like you would your castle.

The difference between programming and other creative activities is that when programming, you have all the raw materials you need in your computer; you don't need to buy any additional canvas, paint, film, yarn, LEGO bricks, or electronic components. When your program is written, it can easily be shared online with the entire world. And though you'll make mistakes when programming, the activity is still a lot of fun.

About This Book

The first part of this book covers basic Python programming concepts, and the second part covers various tasks you can have your computer automate. Each chapter in the second part has project programs for you to study. Here's a brief rundown of what you'll find in each chapter:

Part I: Python Programming Basics

Chapter 1: Python Basics Covers expressions, the most basic type of Python instruction, and how to use the Python interactive shell software to experiment with code.

Chapter 2: Flow Control Explains how to make programs decide which instructions to execute so your code can intelligently respond to different conditions.

Chapter 3: Functions Instructs you on how to define your own functions so that you can organize your code into more manageable chunks.

Chapter 4: Lists Introduces the list data type and explains how to organize data.

Chapter 5: Dictionaries and Structuring Data Introduces the dictionary data type and shows you more powerful ways to organize data.

Chapter 6: Manipulating Strings Covers working with text data (called *strings* in Python).

Part II: Automating Tasks

Chapter 7: Pattern Matching with Regular Expressions Covers how Python can manipulate strings and search for text patterns with regular expressions.

Chapter 8: Reading and Writing Files Explains how your programs can read the contents of text files and save information to files on your hard drive.

Chapter 9: Organizing Files Shows how Python can copy, move, rename, and delete large numbers of files much faster than a human user can. It also explains compressing and decompressing files.

Chapter 10: Debugging Shows how to use Python’s various bug-finding and bug-fixing tools.

Chapter 11: Web Scraping Shows how to write programs that can automatically download web pages and parse them for information. This is called *web scraping*.

Chapter 12: Working with Excel Spreadsheets Covers programmatically manipulating Excel spreadsheets so that you don’t have to read them. This is helpful when the number of documents you have to analyze is in the hundreds or thousands.

Chapter 13: Working with PDF and Word Documents Covers programmatically reading Word and PDF documents.

Chapter 14: Working with CSV Files and JSON Data Continues to explain how to programmatically manipulate documents with CSV and JSON files.

Chapter 15: Keeping Time, Scheduling Tasks, and Launching Programs Explains how time and dates are handled by Python programs and how to schedule your computer to perform tasks at certain times. This chapter also shows how your Python programs can launch non-Python programs.

Chapter 16: Sending Email and Text Messages Explains how to write programs that can send emails and text messages on your behalf.

Chapter 17: Manipulating Images Explains how to programmatically manipulate images such as JPEG or PNG files.

Chapter 18: Controlling the Keyboard and Mouse with GUI Automation Explains how to programmatically control the mouse and keyboard to automate clicks and keypresses.

Downloading and Installing Python

You can download Python for Windows, OS X, and Ubuntu for free from <http://python.org/downloads/>. If you download the latest version from the website’s download page, all of the programs in this book should work.

WARNING

Be sure to download a version of Python 3 (such as 3.4.0). The programs in this book are written to run on Python 3 and may not run correctly, if at all, on Python 2.

You’ll find Python installers for 64-bit and 32-bit computers for each operating system on the download page, so first figure out which installer you need. If you bought your computer in 2007 or later, it is most likely a 64-bit system. Otherwise, you have a 32-bit version, but here’s how to find out for sure:

- On Windows, select **Start ▶ Control Panel ▶ System** and check whether System Type says 64-bit or 32-bit.

- On OS X, go the Apple menu, select **About This Mac** ▶ **More Info** ▶ **System Report** ▶ **Hardware**, and then look at the Processor Name field. If it says Intel Core Solo or Intel Core Duo, you have a 32-bit machine. If it says anything else (including Intel Core 2 Duo), you have a 64-bit machine.
- On Ubuntu Linux, open a Terminal and run the command `uname -m`. A response of `i686` means 32-bit, and `x86_64` means 64-bit.

On Windows, download the Python installer (the filename will end with `.msi`) and double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:

1. Select **Install for All Users** and then click **Next**.
2. Install to the `C:\Python34` folder by clicking **Next**.
3. Click **Next** again to skip the Customize Python section.

On Mac OS X, download the `.dmg` file that's right for your version of OS X and double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:

1. When the DMG package opens in a new window, double-click the `Python.mpkg` file. You may have to enter the administrator password.
2. Click **Continue** through the Welcome section and click **Agree** to accept the license.
3. Select **HD Macintosh** (or whatever name your hard drive has) and click **Install**.

If you're running Ubuntu, you can install Python from the Terminal by following these steps:

1. Open the Terminal window.
2. Enter `sudo apt-get install python3`.
3. Enter `sudo apt-get install idle3`.
4. Enter `sudo apt-get install python3-pip`.

Starting IDLE

While the *Python interpreter* is the software that runs your Python programs, the *interactive development environment (IDLE)* software is where you'll enter your programs, much like a word processor. Let's start IDLE now.

- On Windows 7 or newer, click the Start icon in the lower-left corner of your screen, enter **IDLE** in the search box, and select **IDLE (Python GUI)**.
- On Windows XP, click the **Start** button and then select **Programs** ▶ **Python 3.4** ▶ **IDLE (Python GUI)**.

- On Mac OS X, open the Finder window, click **Applications**, click **Python 3.4**, and then click the IDLE icon.
- On Ubuntu, select **Applications** ▶ **Accessories** ▶ **Terminal** and then enter **idle3**. (You may also be able to click **Applications** at the top of the screen, select **Programming**, and then click **IDLE 3**.)

The Interactive Shell

No matter which operating system you're running, the IDLE window that first appears should be mostly blank except for text that looks something like this:

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64
bit (AMD64)] on win32Type "copyright", "credits" or "license()" for more
information.
```

```
>>>
```

This window is called the *interactive shell*. A shell is a program that lets you type instructions into the computer, much like the Terminal or Command Prompt on OS X and Windows, respectively. Python's interactive shell lets you enter instructions for the Python interpreter software to run. The computer reads the instructions you enter and runs them immediately.

For example, enter the following into the interactive shell next to the >>> prompt:

```
>>> print('Hello world!')
```

After you type that line and press ENTER, the interactive shell should display this in response:

```
>>> print('Hello world!')
Hello world!
```

How to Find Help

Solving programming problems on your own is easier than you might think. If you're not convinced, then let's cause an error on purpose: Enter '42' + 3 into the interactive shell. You don't need to know what this instruction means right now, but the result should look like this:

```
>>> '42' + 3
❶ Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    '42' + 3
❷ TypeError: Can't convert 'int' object to str implicitly
>>>
```

The error message ❷ appeared here because Python couldn't understand your instruction. The traceback part ❶ of the error message shows the specific instruction and line number that Python had trouble with. If you're not sure what to make of a particular error message, search online for the exact error message. Enter “**TypeError: Can't convert 'int' object to str implicitly**” (including the quotes) into your favorite search engine, and you should see tons of links explaining what the error message means and what causes it, as shown in Figure 0-2.



Figure 0-2: The Google results for an error message can be very helpful.

You'll often find that someone else had the same question as you and that some other helpful person has already answered it. No one person can know everything about programming, so an everyday part of any software developer's job is looking up answers to technical questions.

Asking Smart Programming Questions

If you can't find the answer by searching online, try asking people in a web forum such as Stack Overflow (<http://stackoverflow.com/>) or the “learn programming” subreddit at <http://reddit.com/r/learnprogramming/>. But keep in mind there are smart ways to ask programming questions that help others help you. Be sure to read the Frequently Asked Questions sections these websites have about the proper way to post questions.

When asking programming questions, remember to do the following:

- Explain what you are trying to do, not just what you did. This lets your helper know if you are on the wrong track.
- Specify the point at which the error happens. Does it occur at the very start of the program or only after you do a certain action?
- Copy and paste the *entire* error message and your code to <http://pastebin.com/> or <http://gist.github.com/>.

These websites make it easy to share large amounts of code with people over the Web, without the risk of losing any text formatting. You can then put the URL of the posted code in your email or forum post. For example, here some pieces of code I've posted: <http://pastebin.com/SzP2DbFx>/ and <https://gist.github.com/asweigart/6912168>/.

- Explain what you've already tried to do to solve your problem. This tells people you've already put in some work to figure things out on your own.
- List the version of Python you're using. (There are some key differences between version 2 Python interpreters and version 3 Python interpreters.) Also, say which operating system and version you're running.
- If the error came up after you made a change to your code, explain exactly what you changed.
- Say whether you're able to reproduce the error every time you run the program or whether it happens only after you perform certain actions. Explain what those actions are, if so.

Always follow good online etiquette as well. For example, don't post your questions in all caps or make unreasonable demands of the people trying to help you.

Summary

For most people, their computer is just an appliance instead of a tool. But by learning how to program, you'll gain access to one of the most powerful tools of the modern world, and you'll have fun along the way. Programming isn't brain surgery—it's fine for amateurs to experiment and make mistakes.

I love helping people discover Python. I write programming tutorials on my blog at <http://inventwithpython.com/blog/>, and you can contact me with questions at al@inventwithpython.com.

This book will start you off from zero programming knowledge, but you may have questions beyond its scope. Remember that asking effective questions and knowing how to find answers are invaluable tools on your programming journey.

Let's begin!

PART I

PYTHON PROGRAMMING BASICS

1

PYTHON BASICS



The Python programming language has a wide range of syntactical constructions, standard library functions, and interactive development environment features. Fortunately, you can ignore most of that; you just need to learn enough to write some handy little programs.

You will, however, have to learn some basic programming concepts before you can do anything. Like a wizard-in-training, you might think these concepts seem arcane and tedious, but with some knowledge and practice, you'll be able to command your computer like a magic wand to perform incredible feats.

This chapter has a few examples that encourage you to type into the interactive shell, which lets you execute Python instructions one at a time and shows you the results instantly. Using the interactive shell is great for learning what basic Python instructions do, so give it a try as you follow along. You'll remember the things you do much better than the things you only read.

Entering Expressions into the Interactive Shell

You run the interactive shell by launching IDLE, which you installed with Python in the introduction. On Windows, open the Start menu, select **All Programs** ▶ **Python 3.3**, and then select **IDLE (Python GUI)**. On OS X, select **Applications** ▶ **MacPython 3.3** ▶ **IDLE**. On Ubuntu, open a new Terminal window and enter `idle3`.

A window with the `>>>` prompt should appear; that's the interactive shell. Enter `2 + 2` at the prompt to have Python do some simple math.

```
>>> 2 + 2
4
```

The IDLE window should now show some text like this:

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2 + 2
4
>>>
```

In Python, `2 + 2` is called an *expression*, which is the most basic kind of programming instruction in the language. Expressions consist of *values* (such as `2`) and *operators* (such as `+`), and they can always *evaluate* (that is, reduce) down to a single value. That means you can use expressions anywhere in Python code that you could also use a value.

In the previous example, `2 + 2` is evaluated down to a single value, `4`. A single value with no operators is also considered an expression, though it evaluates only to itself, as shown here:

```
>>> 2
2
```

ERRORS ARE OKAY!

Programs will crash if they contain code the computer can't understand, which will cause Python to show an error message. An error message won't break your computer, though, so don't be afraid to make mistakes. A *crash* just means the program stopped running unexpectedly.

If you want to know more about an error message, you can search for the exact message text online to find out more about that specific error. You can also check out the resources at <http://nostarch.com/automatestuff/> to see a list of common Python error messages and their meanings.

There are plenty of other operators you can use in Python expressions, too. For example, Table 1-1 lists all the math operators in Python.

Table 1-1: Math Operators from Highest to Lowest Precedence

Operator	Operation	Example	Evaluates to...
<code>**</code>	Exponent	<code>2 ** 3</code>	8
<code>%</code>	Modulus/remainder	<code>22 % 8</code>	6
<code>//</code>	Integer division/floored quotient	<code>22 // 8</code>	2
<code>/</code>	Division	<code>22 / 8</code>	2.75
<code>*</code>	Multiplication	<code>3 * 5</code>	15
<code>-</code>	Subtraction	<code>5 - 2</code>	3
<code>+</code>	Addition	<code>2 + 2</code>	4

The order of operations (also called *precedence*) of Python math operators is similar to that of mathematics. The `**` operator is evaluated first; the `*`, `/`, `//`, and `%` operators are evaluated next, from left to right; and the `+` and `-` operators are evaluated last (also from left to right). You can use parentheses to override the usual precedence if you need to. Enter the following expressions into the interactive shell:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2      +      2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

In each case, you as the programmer must enter the expression, but Python does the hard part of evaluating it down to a single value. Python will keep evaluating parts of the expression until it becomes a single value, as shown in Figure 1-1.

```
(5 - 1) * ((7 + 1) / (3 - 1))
↓
4 * ((7 + 1) / (3 - 1))
↓
4 * ( 8 ) / (3 - 1)
↓
4 * ( 8 ) / ( 2 )
↓
4 * 4.0
↓
16.0
```

Figure 1-1: Evaluating an expression reduces it to a single value.

These rules for putting operators and values together to form expressions are a fundamental part of Python as a programming language, just like the grammar rules that help us communicate. Here's an example:

This is a grammatically correct English sentence.

This grammatically is sentence not English correct a.

The second line is difficult to parse because it doesn't follow the rules of English. Similarly, if you type in a bad Python instruction, Python won't be able to understand it and will display a `SyntaxError` error message, as shown here:

```
>>> 5 +
      File "<stdin>", line 1
      5 +
      ^
SyntaxError: invalid syntax
>>> 42 + 5 + * 2
      File "<stdin>", line 1
      42 + 5 + * 2
      ^
SyntaxError: invalid syntax
```

You can always test to see whether an instruction works by typing it into the interactive shell. Don't worry about breaking the computer: The worst thing that could happen is that Python responds with an error message. Professional software developers get error messages while writing code all the time.

The Integer, Floating-Point, and String Data Types

Remember that expressions are just values combined with operators, and they always evaluate down to a single value. A *data type* is a category for values, and every value belongs to exactly one data type. The most

common data types in Python are listed in Table 1-2. The values -2 and 30, for example, are said to be *integer* values. The integer (or *int*) data type indicates values that are whole numbers. Numbers with a decimal point, such as 3.14, are called *floating-point numbers* (or *floats*). Note that even though the value 42 is an integer, the value 42.0 would be a floating-point number.

Table 1-2: Common Data Types

Data type	Examples
Integers	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers	-1.25, -1.0, -0.5, 0.0, 0.5, 1.0, 1.25
Strings	'a', 'aa', 'aaa', 'Hello!', '11 cats'

Python programs can also have text values called *strings*, or *sts* (pronounced “stirs”). Always surround your string in single quote ('') characters (as in 'Hello' or 'Goodbye cruel world!') so Python knows where the string begins and ends. You can even have a string with no characters in it, '', called a *blank string*. Strings are explained in greater detail in Chapter 4.

If you ever see the error message `SyntaxError: EOL while scanning string literal`, you probably forgot the final single quote character at the end of the string, such as in this example:

```
>>> 'Hello world!
SyntaxError: EOL while scanning string literal
```

String Concatenation and Replication

The meaning of an operator may change based on the data types of the values next to it. For example, + is the addition operator when it operates on two integers or floating-point values. However, when + is used on two string values, it joins the strings as the *string concatenation* operator. Enter the following into the interactive shell:

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

The expression evaluates down to a single, new string value that combines the text of the two strings. However, if you try to use the + operator on a string and an integer value, Python will not know how to handle this, and it will display an error message.

```
>>> 'Alice' + 42
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    'Alice' + 42
TypeError: Can't convert 'int' object to str implicitly
```

The error message `Can't convert 'int' object to str implicitly` means that Python thought you were trying to concatenate an integer to the string `'Alice'`. Your code will have to explicitly convert the integer to a string, because Python cannot do this automatically. (Converting data types will be explained in “Dissecting Your Program” on page 22 when talking about the `str()`, `int()`, and `float()` functions.)

The `*` operator is used for multiplication when it operates on two integer or floating-point values. But when the `*` operator is used on one string value and one integer value, it becomes the *string replication* operator. Enter a string multiplied by a number into the interactive shell to see this in action.

```
>>> 'Alice' * 5
'AliceAliceAliceAlice'
```

The expression evaluates down to a single string value that repeats the original a number of times equal to the integer value. String replication is a useful trick, but it’s not used as often as string concatenation.

The `*` operator can be used with only two numeric values (for multiplication) or one string value and one integer value (for string replication). Otherwise, Python will just display an error message.

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'Alice' * 5.0
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

It makes sense that Python wouldn’t understand these expressions: You can’t multiply two words, and it’s hard to replicate an arbitrary string a fractional number of times.

Storing Values in Variables

A *variable* is like a box in the computer’s memory where you can store a single value. If you want to use the result of an evaluated expression later in your program, you can save it inside a variable.

Assignment Statements

You’ll store values in variables with an *assignment statement*. An assignment statement consists of a variable name, an equal sign (called the *assignment operator*), and the value to be stored. If you enter the assignment statement `spam = 42`, then a variable named `spam` will have the integer value 42 stored in it.

Think of a variable as a labeled box that a value is placed in, as in Figure 1-2.



Figure 1-2: `spam = 42` is like telling the program,
“The variable `spam` now has the integer value 42 in it.”

For example, enter the following into the interactive shell:

```
❶ >>> spam = 40
>>> spam
40
>>> eggs = 2
❷ >>> spam + eggs
42
>>> spam + eggs + spam
82
❸ >>> spam = spam + 2
>>> spam
42
```

A variable is *initialized* (or created) the first time a value is stored in it ❶. After that, you can use it in expressions with other variables and values ❷. When a variable is assigned a new value ❸, the old value is forgotten, which is why `spam` evaluated to 42 instead of 40 at the end of the example. This is called *overwriting* the variable. Enter the following code into the interactive shell to try overwriting a string:

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

Just like the box in Figure 1-3, the `spam` variable in this example stores '`Hello`' until you replace it with '`Goodbye`'.



Figure 1-3: When a new value is assigned to a variable, the old one is forgotten.

Variable Names

Table 1-3 has examples of legal variable names. You can name a variable anything as long as it obeys the following three rules:

1. It can be only one word.
2. It can use only letters, numbers, and the underscore (_) character.
3. It can't begin with a number.

Table 1-3: Valid and Invalid Variable Names

Valid variable names	Invalid variable names
balance	current-balnce (hyphens are not allowed)
currentBalance	current balance (spaces are not allowed)
current_balance	4account (can't begin with a number)
_spam	42 (can't begin with a number)
SPAM	total_\$um (special characters like \$ are not allowed)
account4	'hello' (special characters like ' are not allowed)

Variable names are case-sensitive, meaning that `spam`, `SPAM`, `Spam`, and `sPaM` are four different variables. It is a Python convention to start your variables with a lowercase letter.

This book uses camelcase for variable names instead of underscores; that is, variables `lookLikeThis` instead of `looking_like_this`. Some experienced programmers may point out that the official Python code style, PEP 8, says that underscores should be used. I unapologetically prefer camelcase and point to “A Foolish Consistency Is the Hobgoblin of Little Minds” in PEP 8 itself:

“Consistency with the style guide is important. But most importantly: know when to be inconsistent—sometimes the style guide just doesn’t apply. When in doubt, use your best judgment.”

A good variable name describes the data it contains. Imagine that you moved to a new house and labeled all of your moving boxes as *Stuff*. You’d never find anything! The variable names `spam`, `eggs`, and `bacon` are used as generic names for the examples in this book and in much of Python’s documentation (inspired by the Monty Python “Spam” sketch), but in your programs, a descriptive name will help make your code more readable.

Your First Program

While the interactive shell is good for running Python instructions one at a time, to write entire Python programs, you’ll type the instructions into the file editor. The *file editor* is similar to text editors such as Notepad or TextMate, but it has some specific features for typing in source code. To open the file editor in IDLE, select **File ▶ New Window**.

The window that appears should contain a cursor awaiting your input, but it’s different from the interactive shell, which runs Python instructions as soon as you press ENTER. The file editor lets you type in many instructions, save the file, and run the program. Here’s how you can tell the difference between the two:

- The interactive shell window will always be the one with the `>>>` prompt.
- The file editor window will not have the `>>>` prompt.

Now it’s time to create your first program! When the file editor window opens, type the following into it:

```
❶ # This program says hello and asks for my name.

❷ print('Hello world!')
    print('What is your name?')      # ask for their name
❸ myName = input()
❹ print('It is good to meet you, ' + myName)
❺ print('The length of your name is:')
    print(len(myName))
```

```
❶ print('What is your age?')      # ask for their age
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

Once you've entered your source code, save it so that you won't have to retype it each time you start IDLE. From the menu at the top of the file editor window, select **File** ▶ **Save As**. In the Save As window, enter **hello.py** in the File Name field and then click **Save**.

You should save your programs every once in a while as you type them. That way, if the computer crashes or you accidentally exit from IDLE, you won't lose the code. As a shortcut, you can press **CTRL-S** on Windows and **⌘-S** on OS X to save your file.

Once you've saved, let's run our program. Select **Run** ▶ **Run Module** or just press the **F5** key. Your program should run in the interactive shell window that appeared when you first started IDLE. Remember, you have to press F5 from the file editor window, not the interactive shell window. Enter your name when your program asks for it. The program's output in the interactive shell should look something like this:

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Hello world!
What is your name?
A1
It is good to meet you, A1
The length of your name is:
2
What is your age?
4
You will be 5 in a year.
>>>
```

When there are no more lines of code to execute, the Python program *terminates*; that is, it stops running. (You can also say that the Python program *exits*.)

You can close the file editor by clicking the X at the top of the window. To reload a saved program, select **File** ▶ **Open** from the menu. Do that now, and in the window that appears, choose **hello.py**, and click the **Open** button. Your previously saved **hello.py** program should open in the file editor window.

Dissecting Your Program

With your new program open in the file editor, let's take a quick tour of the Python instructions it uses by looking at what each line of code does.

Comments

The following line is called a *comment*.

❶ `# This program says hello and asks for my name.`

Python ignores comments, and you can use them to write notes or remind yourself what the code is trying to do. Any text for the rest of the line following a hash mark (#) is part of a comment.

Sometimes, programmers will put a # in front of a line of code to temporarily remove it while testing a program. This is called *commenting out* code, and it can be useful when you're trying to figure out why a program doesn't work. You can remove the # later when you are ready to put the line back in.

Python also ignores the blank line after the comment. You can add as many blank lines to your program as you want. This can make your code easier to read, like paragraphs in a book.

The `print()` Function

The `print()` function displays the string value inside the parentheses on the screen.

❷ `print('Hello world!')`
`print('What is your name?') # ask for their name`

The line `print('Hello world!')` means “Print out the text in the string ‘Hello world!’.” When Python executes this line, you say that Python is *calling* the `print()` function and the string value is being *passed* to the function. A value that is passed to a function call is an *argument*. Notice that the quotes are not printed to the screen. They just mark where the string begins and ends; they are not part of the string value.

NOTE

You can also use this function to put a blank line on the screen; just call `print()` with nothing in between the parentheses.

When writing a function name, the opening and closing parentheses at the end identify it as the name of a function. This is why in this book you'll see `print()` rather than `print`. Chapter 2 describes functions in more detail.

The `input()` Function

The `input()` function waits for the user to type some text on the keyboard and press ENTER.

❸ `myName = input()`

This function call evaluates to a string equal to the user's text, and the previous line of code assigns the `myName` variable to this string value.

You can think of the `input()` function call as an expression that evaluates to whatever string the user typed in. If the user entered 'Al', then the expression would evaluate to `myName = 'Al'`.

Printing the User's Name

The following call to `print()` actually contains the expression 'It is good to meet you, ' + `myName` between the parentheses.

④ `print('It is good to meet you, ' + myName)`

Remember that expressions can always evaluate to a single value. If 'Al' is the value stored in `myName` on the previous line, then this expression evaluates to 'It is good to meet you, Al'. This single string value is then passed to `print()`, which prints it on the screen.

The `len()` Function

You can pass the `len()` function a string value (or a variable containing a string), and the function evaluates to the integer value of the number of characters in that string.

⑤ `print('The length of your name is:')`
`print(len(myName))`

Enter the following into the interactive shell to try this:

```
>>> len('hello')
5
>>> len('My very energetic monster just scarfed nachos.')
46
>>> len('')
0
```

Just like those examples, `len(myName)` evaluates to an integer. It is then passed to `print()` to be displayed on the screen. Notice that `print()` allows you to pass it either integer values or string values. But notice the error that shows up when you type the following into the interactive shell:

```
>>> print('I am ' + 29 + ' years old.')
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print('I am ' + 29 + ' years old.')
TypeError: Can't convert 'int' object to str implicitly
```

The `print()` function isn't causing that error, but rather it's the expression you tried to pass to `print()`. You get the same error message if you type the expression into the interactive shell on its own.

```
>>> 'I am ' + 29 + ' years old.'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    'I am ' + 29 + ' years old.'
TypeError: Can't convert 'int' object to str implicitly
```

Python gives an error because you can use the `+` operator only to add two integers together or concatenate two strings. You can't add an integer to a string because this is ungrammatical in Python. You can fix this by using a string version of the integer instead, as explained in the next section.

The `str()`, `int()`, and `float()` Functions

If you want to concatenate an integer such as `29` with a string to pass to `print()`, you'll need to get the value '`29`', which is the string form of `29`. The `str()` function can be passed an integer value and will evaluate to a string value version of it, as follows:

```
>>> str(29)
'29'
>>> print('I am ' + str(29) + ' years old.')
I am 29 years old.
```

Because `str(29)` evaluates to '`29`', the expression '`I am ' + str(29) + ' years old.'`' evaluates to '`I am ' + '29' + ' years old.'`', which in turn evaluates to '`I am 29 years old.`'. This is the value that is passed to the `print()` function.

The `str()`, `int()`, and `float()` functions will evaluate to the string, integer, and floating-point forms of the value you pass, respectively. Try converting some values in the interactive shell with these functions, and watch what happens.

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

The previous examples call the `str()`, `int()`, and `float()` functions and pass them values of the other data types to obtain a string, integer, or floating-point form of those values.

The `str()` function is handy when you have an integer or float that you want to concatenate to a string. The `int()` function is also helpful if you have a number as a string value that you want to use in some mathematics. For example, the `input()` function always returns a string, even if the user enters a number. Enter `spam = input()` into the interactive shell and enter `101` when it waits for your text.

```
>>> spam = input()
101
>>> spam
'101'
```

The value stored inside `spam` isn't the integer `101` but the string '`101`'. If you want to do math using the value in `spam`, use the `int()` function to get the integer form of `spam` and then store this as the new value in `spam`.

```
>>> spam = int(spam)
>>> spam
101
```

Now you should be able to treat the `spam` variable as an integer instead of a string.

```
>>> spam * 10 / 5
202.0
```

Note that if you pass a value to `int()` that it cannot evaluate as an integer, Python will display an error message.

```
>>> int('99.99')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
>>> int('twelve')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    int('twelve')
ValueError: invalid literal for int() with base 10: 'twelve'
```

The `int()` function is also useful if you need to round a floating-point number down.

```
>>> int(7.7)
7
>>> int(7.7) + 1
8
```

In your program, you used the `int()` and `str()` functions in the last three lines to get a value of the appropriate data type for the code.

```
❶ print('What is your age?') # ask for their age
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

The `myAge` variable contains the value returned from `input()`. Because the `input()` function always returns a string (even if the user typed in a number), you can use the `int(myAge)` code to return an integer value of the string in `myAge`. This integer value is then added to `1` in the expression `int(myAge) + 1`.

The result of this addition is passed to the `str()` function: `str(int(myAge) + 1)`. The string value returned is then concatenated with the strings '`You will be` ' and '`in a year.`' to evaluate to one large string value. This large string is finally passed to `print()` to be displayed on the screen.

Let's say the user enters the string '`4`' for `myAge`. The string '`4`' is converted to an integer, so you can add one to it. The result is `5`. The `str()` function converts the result back to a string, so you can concatenate it with the second string, '`in a year.`', to create the final message. These evaluation steps would look something like Figure 1-4.

TEXT AND NUMBER EQUIVALENCE

Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point.

```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

Python makes this distinction because strings are text, while integers and floats are both numbers.

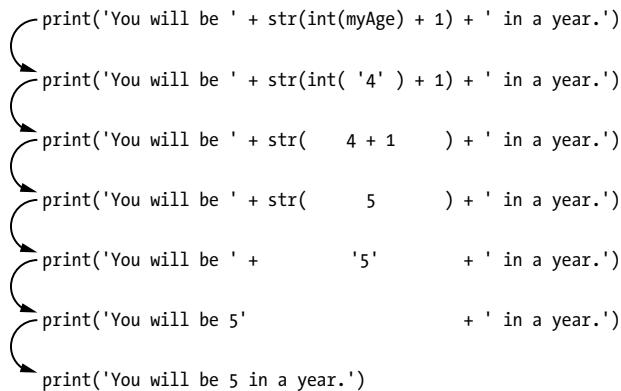
```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
print('You will be ' + str(int( '4' ) + 1) + ' in a year.')
print('You will be ' + str(    4 + 1      ) + ' in a year.')
print('You will be ' + str(      5      ) + ' in a year.')
print('You will be ' +           '5'           + ' in a year.')
print('You will be 5'                      + ' in a year.')
print('You will be 5 in a year.')
```

Figure 1-4: The evaluation steps, if 4 was stored in myAge

Summary

You can compute expressions with a calculator or type string concatenations with a word processor. You can even do string replication easily by copying and pasting text. But expressions, and their component values—operators, variables, and function calls—are the basic building blocks that make programs. Once you know how to handle these elements, you will be able to instruct Python to operate on large amounts of data for you.

It is good to remember the different types of operators (+, -, *, /, //, %, and ** for math operations, and + and * for string operations) and the three data types (integers, floating-point numbers, and strings) introduced in this chapter.

A few different functions were introduced as well. The `print()` and `input()` functions handle simple text output (to the screen) and input (from the keyboard). The `len()` function takes a string and evaluates to an int of the number of characters in the string. The `str()`, `int()`, and `float()` functions will evaluate to the string, integer, or floating-point number form of the value they are passed.

In the next chapter, you will learn how to tell Python to make intelligent decisions about what code to run, what code to skip, and what code to repeat based on the values it has. This is known as *flow control*, and it allows you to write programs that make intelligent decisions.

Practice Questions

1. Which of the following are operators, and which are values?

*
'hello'
-88.8
-
/
+
5

2. Which of the following is a variable, and which is a string?
-

spam
'spam'

3. Name three data types.
4. What is an expression made up of? What do all expressions do?
5. This chapter introduced assignment statements, like `spam = 10`. What is the difference between an expression and a statement?
6. What does the variable `bacon` contain after the following code runs?
-

`bacon = 20`
`bacon + 1`

7. What should the following two expressions evaluate to?
-

`'spam' + 'spamspam'`
`'spam' * 3`

8. Why is `eggs` a valid variable name while `100` is invalid?
9. What three functions can be used to get the integer, floating-point number, or string version of a value?
10. Why does this expression cause an error? How can you fix it?
-

`'I have eaten ' + 99 + ' burritos.'`

Extra credit: Search online for the Python documentation for the `len()` function. It will be on a web page titled “Built-in Functions.” Skim the list of other functions Python has, look up what the `round()` function does, and experiment with it in the interactive shell.

2

FLOW CONTROL



So you know the basics of individual instructions and that a program is just a series of instructions. But the real strength of programming isn't just running (or *executing*) one instruction after another like a weekend errand list. Based on how the expressions evaluate, the program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. *Flow control statements* can decide which Python instructions to execute under which conditions.

These flow control statements directly correspond to the symbols in a flowchart, so I'll provide flowchart versions of the code discussed in this chapter. Figure 2-1 shows a flowchart for what to do if it's raining. Follow the path made by the arrows from Start to End.



Figure 2-1: A flowchart to tell you what to do if it is raining

In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program. Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles. The starting and ending steps are represented with rounded rectangles.

But before you learn about flow control statements, you first need to learn how to represent those *yes* and *no* options, and you need to understand how to write those branching points as Python code. To that end, let's explore Boolean values, comparison operators, and Boolean operators.

Boolean Values

While the integer, floating-point, and string data types have an unlimited number of possible values, the *Boolean* data type has only two values: `True` and `False`. (*Boolean* is capitalized because the data type is named after mathematician George Boole.) When typed as Python code, the Boolean values `True` and `False` lack the quotes you place around strings, and they always start with a capital `T` or `F`, with the rest of the word in lowercase. Enter the following into the interactive shell. (Some of these instructions are intentionally incorrect, and they'll cause error messages to appear.)

```
❶ >>> spam = True
>>> spam
True
❷ >>> true
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    true
NameError: name 'true' is not defined
❸ >>> True = 2 + 2
SyntaxError: assignment to keyword
```

Like any other value, Boolean values are used in expressions and can be stored in variables ❶. If you don't use the proper case ❷ or you try to use `True` and `False` for variable names ❸, Python will give you an error message.

Comparison Operators

Comparison operators compare two values and evaluate down to a single Boolean value. Table 2-1 lists the comparison operators.

Table 2-1: Comparison Operators

Operator	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code><</code>	Less than
<code>></code>	Greater than
<code><=</code>	Less than or equal to
<code>>=</code>	Greater than or equal to

These operators evaluate to `True` or `False` depending on the values you give them. Let's try some operators now, starting with `==` and `!=`.

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

As you might expect, `==` (equal to) evaluates to `True` when the values on both sides are the same, and `!=` (not equal to) evaluates to `True` when the two values are different. The `==` and `!=` operators can actually work with values of any data type.

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
❶ >>> 42 == '42'
False
```

Note that an integer or floating-point value will always be unequal to a string value. The expression `42 == '42'` ❶ evaluates to `False` because Python considers the integer `42` to be different from the string '`42`'.

The `<`, `>`, `<=`, and `>=` operators, on the other hand, work properly only with integer and floating-point values.

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggCount = 42
❶ >>> eggCount <= 42
True
>>> myAge = 29
❷ >>> myAge >= 10
True
```

THE DIFFERENCE BETWEEN THE `==` AND `=` OPERATORS

You might have noticed that the `==` operator (equal to) has two equal signs, while the `=` operator (assignment) has just one equal sign. It's easy to confuse these two operators with each other. Just remember these points:

- The `==` operator (equal to) asks whether two values are the same as each other.
- The `=` operator (assignment) puts the value on the right into the variable on the left.

To help remember which is which, notice that the `==` operator (equal to) consists of two characters, just like the `!=` operator (not equal to) consists of two characters.

You'll often use comparison operators to compare a variable's value to some other value, like in the `eggCount <= 42` ❶ and `myAge >= 10` ❷ examples. (After all, instead of typing `'dog' != 'cat'` in your code, you could have just typed `True`.) You'll see more examples of this later when you learn about flow control statements.

Boolean Operators

The three Boolean operators (`and`, `or`, and `not`) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value. Let's explore these operators in detail, starting with the `and` operator.

Binary Boolean Operators

The `and` and `or` operators always take two Boolean values (or expressions), so they're considered *binary* operators. The `and` operator evaluates an expression to `True` if *both* Boolean values are `True`; otherwise, it evaluates to `False`. Enter some expressions using `and` into the interactive shell to see it in action.

```
>>> True and True
True
>>> True and False
False
```

A *truth table* shows every possible result of a Boolean operator. Table 2-2 is the truth table for the `and` operator.

Table 2-2: The `and` Operator's Truth Table

Expression	Evaluates to...
True and True	True
True and False	False
False and True	False
False and False	False

On the other hand, the `or` operator evaluates an expression to `True` if *either* of the two Boolean values is `True`. If both are `False`, it evaluates to `False`.

```
>>> False or True
True
>>> False or False
False
```

You can see every possible outcome of the `or` operator in its truth table, shown in Table 2-3.

Table 2-3: The or Operator’s Truth Table

Expression	Evaluates to...
True or True	True
True or False	True
False or True	True
False or False	False

The not Operator

Unlike and and or, the not operator operates on only one Boolean value (or expression). The not operator simply evaluates to the opposite Boolean value.

```
>>> not True  
False  
❶ >>> not not not not True  
True
```

Much like using double negatives in speech and writing, you can nest not operators ❶, though there’s never no reason to do this in real programs. Table 2-4 shows the truth table for not.

Table 2-4: The not Operator’s Truth Table

Expression	Evaluates to...
not True	False
not False	True

Mixing Boolean and Comparison Operators

Since the comparison operators evaluate to Boolean values, you can use them in expressions with the Boolean operators.

Recall that the and, or, and not operators are called Boolean operators because they always operate on the Boolean values True and False. While expressions like $4 < 5$ aren’t Boolean values, they are expressions that evaluate down to Boolean values. Try entering some Boolean expressions that use comparison operators into the interactive shell.

```
>>> (4 < 5) and (5 < 6)  
True  
>>> (4 < 5) and (9 < 6)  
False  
>>> (1 == 2) or (2 == 2)  
True
```

The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value. You can think of the computer's evaluation process for $(4 < 5)$ and $(5 < 6)$ as shown in Figure 2-2.

You can also use multiple Boolean operators in an expression, along with the comparison operators.

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```



Figure 2-2: The process of evaluating $(4 < 5)$ and $(5 < 6)$ to True.

The Boolean operators have an order of operations just like the math operators do. After any math and comparison operators evaluate, Python evaluates the `not` operators first, then the `and` operators, and then the `or` operators.

Elements of Flow Control

Flow control statements often start with a part called the *condition*, and all are followed by a block of code called the *clause*. Before you learn about Python's specific flow control statements, I'll cover what a condition and a block are.

Conditions

The Boolean expressions you've seen so far could all be considered conditions, which are the same thing as expressions; *condition* is just a more specific name in the context of flow control statements. Conditions always evaluate down to a Boolean value, True or False. A flow control statement decides what to do based on whether its condition is True or False, and almost every flow control statement uses a condition.

Blocks of Code

Lines of Python code can be grouped together in *blocks*. You can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.

1. Blocks begin when the indentation increases.
2. Blocks can contain other blocks.
3. Blocks end when the indentation decreases to zero or to a containing block's indentation.

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program, shown here:

```
if name == 'Mary':  
    ❶    print('Hello Mary')  
    if password == 'swordfish':  
        ❷    print('Access granted.')  
    else:  
        ❸    print('Wrong password.')
```

The first block of code ❶ starts at the line `print('Hello Mary')` and contains all the lines after it. Inside this block is another block ❷, which has only a single line in it: `print('Access Granted.')`. The third block ❸ is also one line long: `print('Wrong password.')`.

Program Execution

In the previous chapter's `hello.py` program, Python started executing instructions at the top of the program going down, one after another. The *program execution* (or simply, *execution*) is a term for the current instruction being executed. If you print the source code on paper and put your finger on each line as it is executed, you can think of your finger as the program execution.

Not all programs execute by simply going straight down, however. If you use your finger to trace through a program with flow control statements, you'll likely find yourself jumping around the source code based on conditions, and you'll probably skip entire clauses.

Flow Control Statements

Now, let's explore the most important piece of flow control: the statements themselves. The statements represent the diamonds you saw in the flowchart in Figure 2-1, and they are the actual decisions your programs will make.

***if* Statements**

The most common type of flow control statement is the `if` statement. An `if` statement's clause (that is, the block following the `if` statement) will execute if the statement's condition is `True`. The clause is skipped if the condition is `False`.

In plain English, an `if` statement could be read as, “If this condition is `True`, execute the code in the clause.” In Python, an `if` statement consists of the following:

- The `if` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `if` clause)

For example, let's say you have some code that checks to see whether someone's name is Alice. (Pretend `name` was assigned some value earlier.)

```
if name == 'Alice':  
    print('Hi, Alice.')
```

All flow control statements end with a colon and are followed by a new block of code (the clause). This if statement's clause is the block with `print('Hi, Alice.')`. Figure 2-3 shows what a flowchart of this code would look like.



Figure 2-3: The flowchart for an *if* statement

***else* Statements**

An *if* clause can optionally be followed by an *else* statement. The *else* clause is executed only when the *if* statement's condition is `False`. In plain English, an *else* statement could be read as, “If this condition is true, execute this code. Or else, execute that code.” An *else* statement doesn't have a condition, and in code, an *else* statement always consists of the following:

- The `else` keyword
- A colon
- Starting on the next line, an indented block of code (called the *else* clause)

Returning to the Alice example, let's look at some code that uses an *else* statement to offer a different greeting if the person's name isn't Alice.

```
if name == 'Alice':  
    print('Hi, Alice.')
```

```
else:  
    print('Hello, stranger.')
```

Figure 2-4 shows what a flowchart of this code would look like.



Figure 2-4: The flowchart for an `else` statement

***elif* Statements**

While only one of the `if` or `else` clauses will execute, you may have a case where you want one of *many* possible clauses to execute. The `elif` statement is an “`else if`” statement that always follows an `if` or another `elif` statement. It provides another condition that is checked only if any of the previous conditions were `False`. In code, an `elif` statement always consists of the following:

- The `elif` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `elif` clause)

Let's add an `elif` to the name checker to see this statement in action.

```
if name == 'Alice':  
    print('Hi, Alice.')  
elif age < 12:  
    print('You are not Alice, kiddo.')
```

This time, you check the person's age, and the program will tell them something different if they're younger than 12. You can see the flowchart for this in Figure 2-5.



Figure 2-5: The flowchart for an `elif` statement

The `elif` clause executes if `age < 12` is `True` and `name == 'Alice'` is `False`. However, if both of the conditions are `False`, then both of the clauses are skipped. It is *not* guaranteed that at least one of the clauses will be executed. When there is a chain of `elif` statements, only one or none of the clauses will be executed. Once one of the statements' conditions is found to be `True`, the rest of the `elif` clauses are automatically skipped. For example, open a new file editor window and enter the following code, saving it as `vampire.py`:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.)')
```

Here I've added two more `elif` statements to make the name checker greet a person with different answers based on age. Figure 2-6 shows the flowchart for this.

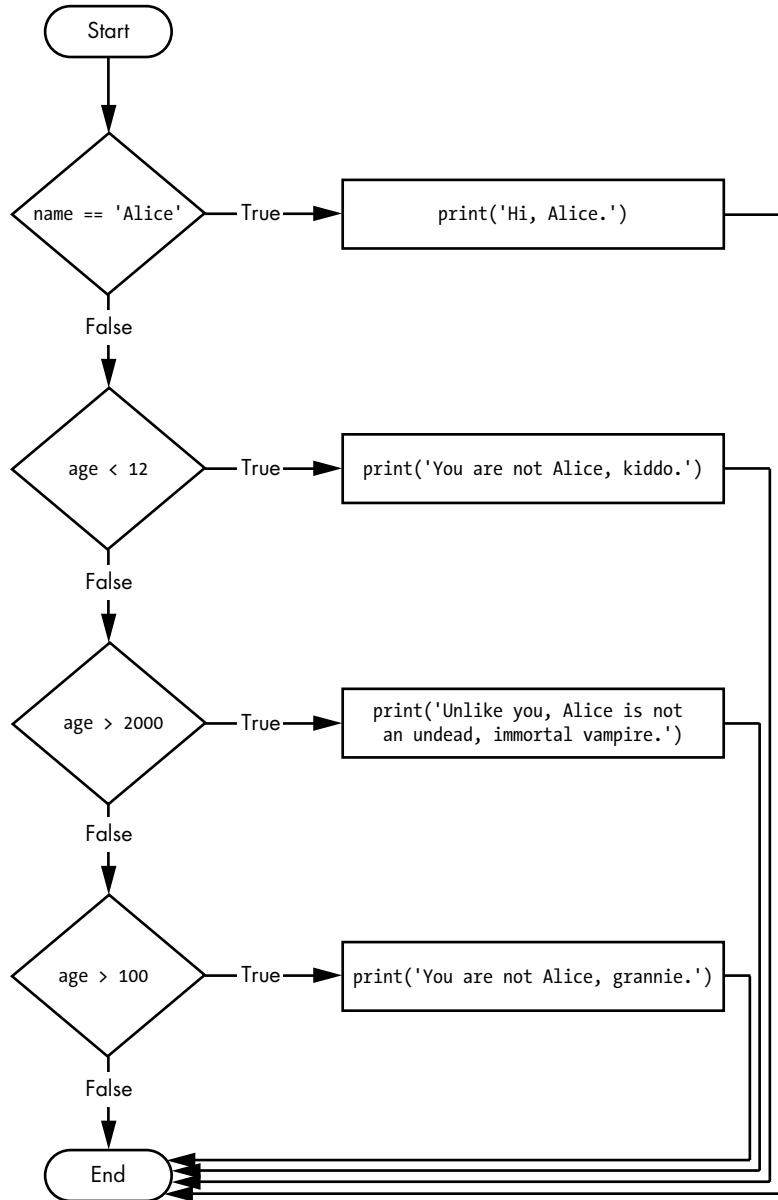


Figure 2-6: The flowchart for multiple `elif` statements in the `vampire.py` program

The order of the `elif` statements does matter, however. Let's rearrange them to introduce a bug. Remember that the rest of the `elif` clauses are automatically skipped once a `True` condition has been found, so if you swap around some of the clauses in `vampire.py`, you run into a problem. Change the code to look like the following, and save it as `vampire2.py`:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
❶ elif age > 100:
    print('You are not Alice, grannie.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')

---


```

Say the `age` variable contains the value `3000` before this code is executed. You might expect the code to print the string '`Unlike you, Alice is not an undead, immortal vampire.`'. However, because the `age > 100` condition is `True` (after all, `3000` is greater than `100`) ❶, the string '`You are not Alice, grannie.`' is printed, and the rest of the `elif` statements are automatically skipped. Remember, at most only one of the clauses will be executed, and for `elif` statements, the order matters!

Figure 2-7 shows the flowchart for the previous code. Notice how the diamonds for `age > 100` and `age > 2000` are swapped.

Optionally, you can have an `else` statement after the last `elif` statement. In that case, it is guaranteed that at least one (and only one) of the clauses will be executed. If the conditions in every `if` and `elif` statement are `False`, then the `else` clause is executed. For example, let's re-create the Alice program to use `if`, `elif`, and `else` clauses.

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')

---


```

Figure 2-8 shows the flowchart for this new code, which we'll save as `littleKid.py`.

In plain English, this type of flow control structure would be, “If the first condition is true, do this. Else, if the second condition is true, do that. Otherwise, do something else.” When you use all three of these statements together, remember these rules about how to order them to avoid bugs like the one in Figure 2-7. First, there is always exactly one `if` statement. Any `elif` statements you need should follow the `if` statement. Second, if you want to be sure that at least one clause is executed, close the structure with an `else` statement.



Figure 2-7: The flowchart for the `vampire2.py` program. The crossed-out path will logically never happen, because if `age` were greater than 2000, it would have already been greater than 100.



Figure 2-8: Flowchart for the previous `littleKid.py` program

***while* Loop Statements**

You can make a block of code execute over and over again with a `while` statement. The code in a `while` clause will be executed as long as the `while` statement's condition is `True`. In code, a `while` statement always consists of the following:

- The `while` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `while` clause)

You can see that a `while` statement looks similar to an `if` statement. The difference is in how they behave. At the end of an `if` clause, the program execution continues after the `if` statement. But at the end of a `while` clause, the program execution jumps back to the start of the `while` statement. The `while` clause is often called the *while loop* or just the *loop*.

Let's look at an `if` statement and a `while` loop that use the same condition and take the same actions based on that condition. Here is the code with an `if` statement:

```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Here is the code with a `while` statement:

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

These statements are similar—both `if` and `while` check the value of `spam`, and if it's less than five, they print a message. But when you run these two code snippets, something very different happens for each one. For the `if` statement, the output is simply "Hello, world.". But for the `while` statement, it's "Hello, world." repeated five times! Take a look at the flowcharts for these two pieces of code, Figures 2-9 and 2-10, to see why this happens.

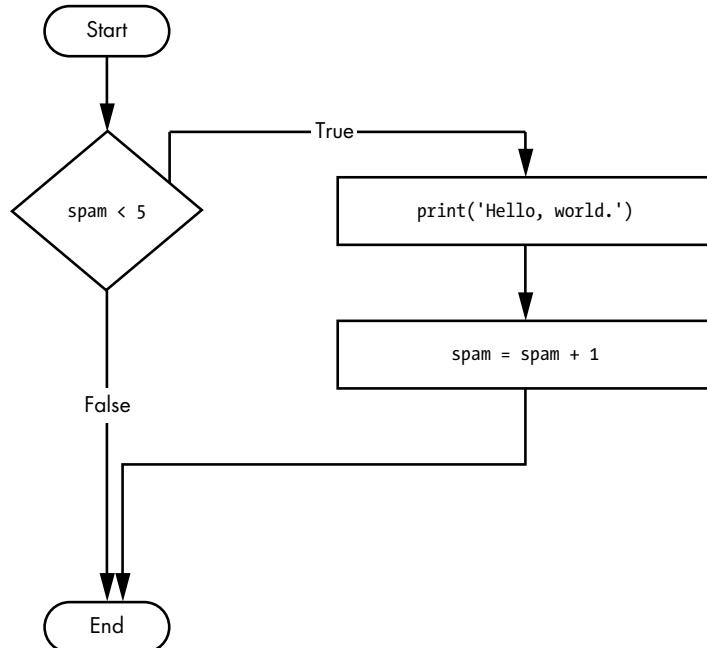


Figure 2-9: The flowchart for the `if` statement code



Figure 2-10: The flowchart for the `while` statement code

The code with the `if` statement checks the condition, and it prints `Hello, world.` only once if that condition is true. The code with the `while` loop, on the other hand, will print it five times. It stops after five prints because the integer in `spam` is incremented by one at the end of each loop iteration, which means that the loop will execute five times before `spam < 5` is `False`.

In the `while` loop, the condition is always checked at the start of each *iteration* (that is, each time the loop is executed). If the condition is `True`, then the clause is executed, and afterward, the condition is checked again. The first time the condition is found to be `False`, the `while` clause is skipped.

An Annoying `while` Loop

Here's a small example program that will keep asking you to type, literally, `your name`. Select **File ▶ New Window** to open a new file editor window, enter the following code, and save the file as `yourName.py`:

```

❶ name = ''
❷ while name != 'your name':
    print('Please type your name.')
❸     name = input()
❹ print('Thank you!')

```

First, the program sets the `name` variable ❶ to an empty string. This is so that the `name != 'your name'` condition will evaluate to `True` and the program execution will enter the `while` loop's clause ❷.

The code inside this clause asks the user to type their name, which is assigned to the `name` variable ❸. Since this is the last line of the block, the execution moves back to the start of the `while` loop and reevaluates the condition. If the value in `name` is *not equal* to the string '`your name`', then the condition is `True`, and the execution enters the `while` clause again.

But once the user types `your name`, the condition of the `while` loop will be '`your name`' `!=` '`your name`', which evaluates to `False`. The condition is now `False`, and instead of the program execution reentering the `while` loop's clause, it skips past it and continues running the rest of the program ❹. Figure 2-11 shows a flowchart for the `yourName.py` program.



Figure 2-11: A flowchart of the `yourName.py` program

Now, let's see `yourName.py` in action. Press **F5** to run it, and enter something other than `your name` a few times before you give the program what it wants.

```
Please type your name.  
A1  
Please type your name.  
Albert
```

```
Please type your name.  
%#@#%^(&!!!  
Please type your name.  
your name  
Thank you!
```

If you never enter `your name`, then the `while` loop's condition will never be `False`, and the program will just keep asking forever. Here, the `input()` call lets the user enter the right string to make the program move on. In other programs, the condition might never actually change, and that can be a problem. Let's look at how you can break out of a `while` loop.

break Statements

There is a shortcut to getting the program execution to break out of a `while` loop's clause early. If the execution reaches a `break` statement, it immediately exits the `while` loop's clause. In code, a `break` statement simply contains the `break` keyword.

Pretty simple, right? Here's a program that does the same thing as the previous program, but it uses a `break` statement to escape the loop. Enter the following code, and save the file as `yourName2.py`:

```
❶ while True:  
    print('Please type your name.')  
❷    name = input()  
❸    if name == 'your name':  
❹        break  
❺ print('Thank you!')
```

The first line ❶ creates an *infinite loop*; it is a `while` loop whose condition is always `True`. (The expression `True`, after all, always evaluates down to the value `True`.) The program execution will always enter the loop and will exit it only when a `break` statement is executed. (An infinite loop that *never* exits is a common programming bug.)

Just like before, this program asks the user to type `your name` ❷. Now, however, while the execution is still inside the `while` loop, an `if` statement gets executed ❸ to check whether `name` is equal to `your name`. If this condition is `True`, the `break` statement is run ❹, and the execution moves out of the loop to `print('Thank you!')` ❺. Otherwise, the `if` statement's clause with the `break` statement is skipped, which puts the execution at the end of the `while` loop. At this point, the program execution jumps back to the start of the `while` statement ❶ to recheck the condition. Since this condition is merely the `True` Boolean value, the execution enters the loop to ask the user to type `your name` again. See Figure 2-12 for the flowchart of this program.

Run `yourName2.py`, and enter the same text you entered for `yourName.py`. The rewritten program should respond in the same way as the original.



Figure 2-12: The flowchart for the `yourName2.py` program with an infinite loop. Note that the X path will logically never happen because the loop condition is always True.

continue Statements

Like `break` statements, `continue` statements are used inside loops. When the program execution reaches a `continue` statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition. (This is also what happens when the execution reaches the end of the loop.)

TRAPPED IN AN INFINITE LOOP?

If you ever run a program that has a bug causing it to get stuck in an infinite loop, press **CTRL-C**. This will send a `KeyboardInterrupt` error to your program and cause it to stop immediately. To try it, create a simple infinite loop in the file editor, and save it as `inifiniteloop.py`.

```
while True:  
    print('Hello world!')
```

When you run this program, it will print `Hello world!` to the screen forever, because the `while` statement's condition is always `True`. In IDLE's interactive shell window, there are only two ways to stop this program: press **CTRL-C** or select **Shell > Restart Shell** from the menu. **CTRL-C** is handy if you ever want to terminate your program immediately, even if it's not stuck in an infinite loop.

Let's use `continue` to write a program that asks for a name and password. Enter the following code into a new file editor window and save the program as `swordfish.py`.

```
while True:  
    print('Who are you?')  
    name = input()  
❶    if name != 'Joe':  
❷        continue  
    print('Hello, Joe. What is the password? (It is a fish.)')  
❸    password = input()  
    if password == 'swordfish':  
❹        break  
❺    print('Access granted.')
```

If the user enters any name besides `Joe` ❶, the `continue` statement ❷ causes the program execution to jump back to the start of the loop. When it reevaluates the condition, the execution will always enter the loop, since the condition is simply the value `True`. Once they make it past that `if` statement, the user is asked for a password ❸. If the password entered is `swordfish`, then the `break` statement ❹ is run, and the execution jumps out of the `while` loop to print `Access granted` ❺. Otherwise, the execution continues to the end of the `while` loop, where it then jumps back to the start of the loop. See Figure 2-13 for this program's flowchart.



Figure 2-13: A flowchart for `swordfish.py`. The X path will logically never happen because the loop condition is always True.

"TRUTHY" AND "FALSEY" VALUES

There are some values in other data types that conditions will consider equivalent to True and False. When used in conditions, 0, 0.0, and '' (the empty string) are considered False, while all other values are considered True. For example, look at the following program:

```
name = ''
while not name:1
    print('Enter your name:')
    name = input()
print('How many guests will you have?')
numOfGuests = int(input())
if numOfGuests:2
    print('Be sure to have enough room for all your guests.')3
print('Done')
```

If the user enters a blank string for name, then the while statement's condition will be True **1**, and the program continues to ask for a name. If the value for numOfGuests is not 0 **2**, then the condition is considered to be True, and the program will print a reminder for the user **3**.

You could have typed not name != '' instead of not name, and numOfGuests != 0 instead of numOfGuests, but using the truthy and falsey values can make your code easier to read.

Run this program and give it some input. Until you claim to be Joe, it shouldn't ask for a password, and once you enter the correct password, it should exit.

```
Who are you?
I'm fine, thanks. Who are you?
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
Mary
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
swordfish
Access granted.
```

for Loops and the range() Function

The while loop keeps looping while its condition is True (which is the reason for its name), but what if you want to execute a block of code only a certain number of times? You can do this with a for loop statement and the range() function.

In code, a `for` statement looks something like `for i in range(5):` and always includes the following:

- The `for` keyword
- A variable name
- The `in` keyword
- A call to the `range()` method with up to three integers passed to it
- A colon
- Starting on the next line, an indented block of code (called the `for` clause)

Let's create a new program called `fiveTimes.py` to help you see a `for` loop in action.

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

The code in the `for` loop's clause is run five times. The first time it is run, the variable `i` is set to 0. The `print()` call in the clause will print `Jimmy Five Times (0)`. After Python finishes an iteration through all the code inside the `for` loop's clause, the execution goes back to the top of the loop, and the `for` statement increments `i` by one. This is why `range(5)` results in five iterations through the clause, with `i` being set to 0, then 1, then 2, then 3, and then 4. The variable `i` will go up to, but will not include, the integer passed to `range()`. Figure 2-14 shows a flowchart for the `fiveTimes.py` program.



Figure 2-14: The flowchart for `fiveTimes.py`

When you run this program, it should print `Jimmy Five Times` followed by the value of `i` five times before leaving the `for` loop.

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

NOTE

You can use `break` and `continue` statements inside `for` loops as well. The `continue` statement will continue to the next value of the `for` loop's counter, as if the program execution had reached the end of the loop and returned to the start. In fact, you can use `continue` and `break` statements only inside `while` and `for` loops. If you try to use these statements elsewhere, Python will give you an error.

As another `for` loop example, consider this story about the mathematician Karl Friedrich Gauss. When Gauss was a boy, a teacher wanted to give the class some busywork. The teacher told them to add up all the numbers from 0 to 100. Young Gauss came up with a clever trick to figure out the answer in a few seconds, but you can write a Python program with a `for` loop to do this calculation for you.

```
❶ total = 0
❷ for num in range(101):
❸     total = total + num
❹ print(total)
```

The result should be 5,050. When the program first starts, the `total` variable is set to 0 ❶. The `for` loop ❷ then executes `total = total + num` ❸ 100 times. By the time the loop has finished all of its 100 iterations, every integer from 0 to 100 will have been added to `total`. At this point, `total` is printed to the screen ❹. Even on the slowest computers, this program takes less than a second to complete.

(Young Gauss figured out that there were 50 pairs of numbers that added up to 100: 1 + 99, 2 + 98, 3 + 97, and so on, until 49 + 51. Since 50×100 is 5,000, when you add that middle 50, the sum of all the numbers from 0 to 100 is 5,050. Clever kid!)

An Equivalent `while` Loop

You can actually use a `while` loop to do the same thing as a `for` loop; `for` loops are just more concise. Let's rewrite `fiveTimes.py` to use a `while` loop equivalent of a `for` loop.

```
print('My name is')
i = 0
while i < 5:
    print('Jimmy Five Times (' + str(i) + ')')
    i = i + 1
```

If you run this program, the output should look the same as the *fiveTimes.py* program, which uses a `for` loop.

The Starting, Stopping, and Stepping Arguments to `range()`

Some functions can be called with multiple arguments separated by a comma, and `range()` is one of them. This lets you change the integer passed to `range()` to follow any sequence of integers, including starting at a number other than zero.

```
for i in range(12, 16):
    print(i)
```

The first argument will be where the `for` loop's variable starts, and the second argument will be up to, but not including, the number to stop at.

```
12
13
14
15
```

The `range()` function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the *step argument*. The step is the amount that the variable is increased by after each iteration.

```
for i in range(0, 10, 2):
    print(i)
```

So calling `range(0, 10, 2)` will count from zero to eight by intervals of two.

```
0
2
4
6
8
```

The `range()` function is flexible in the sequence of numbers it produces for `for` loops. For example (I never apologize for my puns), you can even use a negative number for the step argument to make the `for` loop count down instead of up.

```
for i in range(5, -1, -1):
    print(i)
```

Running a `for` loop to print `i` with `range(5, -1, -1)` should print from five down to zero.

```
5
4
```

```
3  
2  
1  
0
```

Importing Modules

All Python programs can call a basic set of functions called *built-in functions*, including the `print()`, `input()`, and `len()` functions you've seen before. Python also comes with a set of modules called the *standard library*. Each module is a Python program that contains a related group of functions that can be embedded in your programs. For example, the `math` module has mathematics-related functions, the `random` module has random number-related functions, and so on.

Before you can use the functions in a module, you must import the module with an `import` statement. In code, an `import` statement consists of the following:

- The `import` keyword
- The name of the module
- Optionally, more module names, as long as they are separated by commas

Once you import a module, you can use all the cool functions of that module. Let's give it a try with the `random` module, which will give us access to the `random.randint()` function.

Enter this code into the file editor, and save it as `printRandom.py`:

```
import random  
for i in range(5):  
    print(random.randint(1, 10))
```

When you run this program, the output will look something like this:

```
4  
1  
8  
4  
1
```

The `random.randint()` function call evaluates to a random integer value between the two integers that you pass it. Since `randint()` is in the `random` module, you must first type `random.` in front of the function name to tell Python to look for this function inside the `random` module.

Here's an example of an `import` statement that imports four different modules:

```
import random, sys, os, math
```

Now we can use any of the functions in these four modules. We'll learn more about them later in the book.

from import Statements

An alternative form of the `import` statement is composed of the `from` keyword, followed by the module name, the `import` keyword, and a star; for example, `from random import *`.

With this form of `import` statement, calls to functions in `random` will not need the `random.` prefix. However, using the full name makes for more readable code, so it is better to use the normal form of the `import` statement.

Ending a Program Early with `sys.exit()`

The last flow control concept to cover is how to terminate the program. This always happens if the program execution reaches the bottom of the instructions. However, you can cause the program to terminate, or exit, by calling the `sys.exit()` function. Since this function is in the `sys` module, you have to import `sys` before your program can use it.

Open a new file editor window and enter the following code, saving it as `exitExample.py`:

```
import sys

while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

Run this program in IDLE. This program has an infinite loop with no `break` statement inside. The only way this program will end is if the user enters `exit`, causing `sys.exit()` to be called. When `response` is equal to `exit`, the program ends. Since the `response` variable is set by the `input()` function, the user must enter `exit` in order to stop the program.

Summary

By using expressions that evaluate to `True` or `False` (also called *conditions*), you can write programs that make decisions on what code to execute and what code to skip. You can also execute code over and over again in a loop while a certain condition evaluates to `True`. The `break` and `continue` statements are useful if you need to exit a loop or jump back to the start.

These flow control statements will let you write much more intelligent programs. There's another type of flow control that you can achieve by writing your own functions, which is the topic of the next chapter.

Practice Questions

1. What are the two values of the Boolean data type? How do you write them?
2. What are the three Boolean operators?
3. Write out the truth tables of each Boolean operator (that is, every possible combination of Boolean values for the operator and what they evaluate to).
4. What do the following expressions evaluate to?

```
(5 > 4) and (3 == 5)
not (5 > 4)
(5 > 4) or (3 == 5)
not ((5 > 4) or (3 == 5))
(True and True) and (True == False)
(not False) or (not True)
```

5. What are the six comparison operators?
6. What is the difference between the equal to operator and the assignment operator?
7. Explain what a condition is and where you would use one.
8. Identify the three blocks in this code:

```
spam = 0
if spam == 10:
    print('eggs')
    if spam > 5:
        print('bacon')
    else:
        print('ham')
    print('spam')
print('spam')
```

9. Write code that prints `Hello` if `1` is stored in `spam`, prints `Howdy` if `2` is stored in `spam`, and prints `Greetings!` if anything else is stored in `spam`.
10. What can you press if your program is stuck in an infinite loop?
11. What is the difference between `break` and `continue`?
12. What is the difference between `range(10)`, `range(0, 10)`, and `range(0, 10, 1)` in a `for` loop?
13. Write a short program that prints the numbers `1` to `10` using a `for` loop. Then write an equivalent program that prints the numbers `1` to `10` using a `while` loop.
14. If you had a function named `bacon()` inside a module named `spam`, how would you call it after importing `spam`?

Extra credit: Look up the `round()` and `abs()` functions on the Internet, and find out what they do. Experiment with them in the interactive shell.

3

FUNCTIONS



You're already familiar with the `print()`, `input()`, and `len()` functions from the previous chapters. Python provides several built-in functions like these, but you can also write your own functions. A *function* is like a mini-program within a program.

To better understand how functions work, let's create one. Type this program into the file editor and save it as `helloFunc.py`:

```
❶ def hello():
❷     print('Howdy!')
      print('Howdy!!!')
      print('Hello there.')  
  
❸ hello()  
    hello()  
    hello()
```

The first line is a `def` statement ❶, which defines a function named `hello()`. The code in the block that follows the `def` statement ❷ is the body of the function. This code is executed when the function is called, not when the function is first defined.

The `hello()` lines after the function ❸ are function calls. In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses. When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there. When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.

Since this program calls `hello()` three times, the code in the `hello()` function is executed three times. When you run this program, the output looks like this:

```
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

A major purpose of functions is to group code that gets executed multiple times. Without a function defined, you would have to copy and paste this code each time, and the program would look like this:

```
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.'
```

In general, you always want to avoid duplicating code, because if you ever decide to update the code—if, for example, you find a bug you need to fix—you'll have to remember to change the code everywhere you copied it.

As you get more programming experience, you'll often find yourself *deduplicating* code, which means getting rid of duplicated or copy-and-pasted code. Deduplication makes your programs shorter, easier to read, and easier to update.

def Statements with Parameters

When you call the `print()` or `len()` function, you pass in values, called *arguments* in this context, by typing them between the parentheses. You can also define your own functions that accept arguments. Type this example into the file editor and save it as `helloFunc2.py`:

```
❶ def hello(name):
❷     print('Hello ' + name)

❸ hello('Alice')
    hello('Bob')
```

When you run this program, the output looks like this:

```
Hello Alice
Hello Bob
```

The definition of the `hello()` function in this program has a parameter called `name` ❶. A *parameter* is a variable that an argument is stored in when a function is called. The first time the `hello()` function is called, it's with the argument 'Alice' ❸. The program execution enters the function, and the variable `name` is automatically set to 'Alice', which is what gets printed by the `print()` statement ❷.

One special thing to note about parameters is that the value stored in a parameter is forgotten when the function returns. For example, if you added `print(name)` after `hello('Bob')` in the previous program, the program would give you a `NameError` because there is no variable named `name`. This variable was destroyed after the function call `hello('Bob')` had returned, so `print(name)` would refer to a `name` variable that does not exist.

This is similar to how a program's variables are forgotten when the program terminates. I'll talk more about why that happens later in the chapter, when I discuss what a function's local scope is.

Return Values and return Statements

When you call the `len()` function and pass it an argument such as 'Hello', the function call evaluates to the integer value 5, which is the length of the string you passed it. In general, the value that a function call evaluates to is called the *return value* of the function.

When creating a function using the `def` statement, you can specify what the return value should be with a `return` statement. A `return` statement consists of the following:

- The `return` keyword
- The value or expression that the function should return

When an expression is used with a `return` statement, the return value is what this expression evaluates to. For example, the following program defines a function that returns a different string depending on what number it is passed as an argument. Type this code into the file editor and save it as `magic8Ball.py`:

```
❶ import random

❷ def getAnswer(answerNumber):
❸     if answerNumber == 1:
        return 'It is certain'
    elif answerNumber == 2:
        return 'It is decidedly so'
    elif answerNumber == 3:
        return 'Yes'
    elif answerNumber == 4:
        return 'Reply hazy try again'
    elif answerNumber == 5:
        return 'Ask again later'
    elif answerNumber == 6:
        return 'Concentrate and ask again'
    elif answerNumber == 7:
        return 'My reply is no'
    elif answerNumber == 8:
        return 'Outlook not so good'
    elif answerNumber == 9:
        return 'Very doubtful'

❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)
```

When this program starts, Python first imports the `random` module ❶. Then the `getAnswer()` function is defined ❷. Because the function is being defined (and not called), the execution skips over the code in it. Next, the `random.randint()` function is called with two arguments, `1` and `9` ❹. It evaluates to a random integer between `1` and `9` (including `1` and `9` themselves), and this value is stored in a variable named `r`.

The `getAnswer()` function is called with `r` as the argument ❺. The program execution moves to the top of the `getAnswer()` function ❻, and the value `r` is stored in a parameter named `answerNumber`. Then, depending on this value in `answerNumber`, the function returns one of many possible string values. The program execution returns to the line at the bottom of the program that originally called `getAnswer()` ❺. The returned string is assigned to a variable named `fortune`, which then gets passed to a `print()` call ❻ and is printed to the screen.

Note that since you can pass return values as an argument to another function call, you could shorten these three lines:

```
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

to this single equivalent line:

```
print(getAnswer(random.randint(1, 9)))
```

Remember, expressions are composed of values and operators. A function call can be used in an expression because it evaluates to its return value.

The None Value

In Python there is a value called `None`, which represents the absence of a value. `None` is the only value of the `NoneType` data type. (Other programming languages might call this value `null`, `nil`, or `undefined`.) Just like the Boolean `True` and `False` values, `None` must be typed with a capital `N`.

This value-without-a-value can be helpful when you need to store something that won't be confused for a real value in a variable. One place where `None` is used is as the return value of `print()`. The `print()` function displays text on the screen, but it doesn't need to return anything in the same way `len()` or `input()` does. But since all function calls need to evaluate to a return value, `print()` returns `None`. To see this in action, enter the following into the interactive shell:

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

Behind the scenes, Python adds `return None` to the end of any function definition with no `return` statement. This is similar to how a `while` or `for` loop implicitly ends with a `continue` statement. Also, if you use a `return` statement without a value (that is, just the `return` keyword by itself), then `None` is returned.

Keyword Arguments and `print()`

Most arguments are identified by their position in the function call. For example, `random.randint(1, 10)` is different from `random.randint(10, 1)`. The function call `random.randint(1, 10)` will return a random integer between 1 and 10, because the first argument is the low end of the range and the second argument is the high end (while `random.randint(10, 1)` causes an error).

However, *keyword arguments* are identified by the keyword put before them in the function call. Keyword arguments are often used for optional parameters. For example, the `print()` function has the optional parameters `end` and `sep` to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively.

If you ran the following program:

```
print('Hello')
print('World')
```

the output would look like this:

```
Hello
World
```

The two strings appear on separate lines because the `print()` function automatically adds a newline character to the end of the string it is passed. However, you can set the `end` keyword argument to change this to a different string. For example, if the program were this:

```
print('Hello', end='')
print('World')
```

the output would look like this:

```
HelloWorld
```

The output is printed on a single line because there is no longer a newline printed after 'Hello'. Instead, the blank string is printed. This is useful if you need to disable the newline that gets added to the end of every `print()` function call.

Similarly, when you pass multiple string values to `print()`, the function will automatically separate them with a single space. Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

But you could replace the default separating string by passing the `sep` keyword argument. Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

You can add keyword arguments to the functions you write as well, but first you'll have to learn about the list and dictionary data types in the next two chapters. For now, just know that some functions have optional keyword arguments that can be specified when the function is called.

Local and Global Scope

Parameters and variables that are assigned in a called function are said to exist in that function's *local scope*. Variables that are assigned outside all functions are said to exist in the *global scope*. A variable that exists in a local scope is called a *local variable*, while a variable that exists in the global scope is called a *global variable*. A variable must be one or the other; it cannot be both local and global.

Think of a *scope* as a container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten. There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time you ran your program, the variables would remember their values from the last time you ran it.

A local scope is created whenever a function is called. Any variables assigned in this function exist within the local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you call this function, the local variables will not remember the values stored in them from the last time the function was called.

Scopes matter for several reasons:

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named `spam` and a global variable also named `spam`.

The reason Python has different scopes instead of just making everything a global variable is so that when variables are modified by the code in a particular call to a function, the function interacts with the rest of the program only through its parameters and the return value. This narrows down the list code lines that may be causing a bug. If your program contained nothing but global variables and had a bug because of a variable being set to a bad value, then it would be hard to track down where this bad value was set. It could have been set from anywhere in the program—and your program could be hundreds or thousands of lines long! But if the bug is because of a local variable with a bad value, you know that only the code in that one function could have set it incorrectly.

While using global variables in small programs is fine, it is a bad habit to rely on global variables as your programs get larger and larger.

Local Variables Cannot Be Used in the Global Scope

Consider this program, which will cause an error when you run it:

```
def spam():
    eggs = 31337
```

```
spam()  
print(eggs)
```

If you run this program, the output will look like this:

```
Traceback (most recent call last):  
  File "C:/test3784.py", line 4, in <module>  
    print(eggs)  
NameError: name 'eggs' is not defined
```

The error happens because the eggs variable exists only in the local scope created when `spam()` is called. Once the program execution returns from `spam`, that local scope is destroyed, and there is no longer a variable named eggs. So when your program tries to run `print(eggs)`, Python gives you an error saying that eggs is not defined. This makes sense if you think about it; when the program execution is in the global scope, no local scopes exist, so there can't be any local variables. This is why only global variables can be used in the global scope.

Local Scopes Cannot Use Variables in Other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:

```
def spam():  
❶    eggs = 99  
❷    bacon()  
❸    print(eggs)  
  
def bacon():  
    ham = 101  
❹    eggs = 0  
  
❺ spam()
```

When the program starts, the `spam()` function is called ❺, and a local scope is created. The local variable `eggs` ❶ is set to 99. Then the `bacon()` function is called ❷, and a second local scope is created. Multiple local scopes can exist at the same time. In this new local scope, the local variable `ham` is set to 101, and a local variable `eggs`—which is different from the one in `spam()`'s local scope—is also created ❹ and set to 0.

When `bacon()` returns, the local scope for that call is destroyed. The program execution continues in the `spam()` function to print the value of `eggs` ❸, and since the local scope for the call to `spam()` still exists here, the `eggs` variable is set to 99. This is what the program prints.

The upshot is that local variables in one function are completely separate from the local variables in another function.

Global Variables Can Be Read from a Local Scope

Consider the following program:

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

Since there is no parameter named eggs or any code that assigns eggs a value in the `spam()` function, when eggs is used in `spam()`, Python considers it a reference to the global variable eggs. This is why 42 is printed when the previous program is run.

Local and Global Variables with the Same Name

To simplify your life, avoid using local variables that have the same name as a global variable or another local variable. But technically, it's perfectly legal to do so in Python. To see what happens, type the following code into the file editor and save it as `sameName.py`:

```
def spam():
❶    eggs = 'spam local'
    print(eggs)    # prints 'spam local'

def bacon():
❷    eggs = 'bacon local'
    print(eggs)    # prints 'bacon local'
    spam()
    print(eggs)    # prints 'bacon local'

❸ eggs = 'global'
bacon()
print(eggs)        # prints 'global'
```

When you run this program, it outputs the following:

```
bacon local
spam local
bacon local
global
```

There are actually three different variables in this program, but confusingly they are all named eggs. The variables are as follows:

- ❶ A variable named eggs that exists in a local scope when `spam()` is called.
- ❷ A variable named eggs that exists in a local scope when `bacon()` is called.
- ❸ A variable named eggs that exists in the global scope.

Since these three separate variables all have the same name, it can be confusing to keep track of which one is being used at any given time. This is why you should avoid using the same variable name in different scopes.

The global Statement

If you need to modify a global variable from within a function, use the `global` statement. If you have a line such as `global eggs` at the top of a function, it tells Python, “In this function, `eggs` refers to the global variable, so don’t create a local variable with this name.” For example, type the following code into the file editor and save it as `sameName2.py`:

```
def spam():
❶    global eggs
❷    eggs = 'spam'

eggs = 'global'
spam()
print(eggs)
```

When you run this program, the final `print()` call will output this:

```
spam
```

Because `eggs` is declared `global` at the top of `spam()` ❶, when `eggs` is set to '`spam`' ❷, this assignment is done to the globally scoped `spam`. No local `spam` variable is created.

There are four rules to tell whether a variable is in a local scope or global scope:

1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
2. If there is a `global` statement for that variable in a function, it is a global variable.
3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
4. But if the variable is not used in an assignment statement, it is a global variable.

To get a better feel for these rules, here’s an example program. Type the following code into the file editor and save it as `sameName3.py`:

```
def spam():
❶    global eggs
    eggs = 'spam' # this is the global

def bacon():
❷    eggs = 'bacon' # this is a local
```

```
def ham():
❸     print(eggs) # this is the global

eggs = 42 # this is the global
spam()
print(eggs)
```

In the `spam()` function, `eggs` is the global `eggs` variable, because there's a global statement for `eggs` at the beginning of the function ❶. In `bacon()`, `eggs` is a local variable, because there's an assignment statement for it in that function ❷. In `ham()` ❸, `eggs` is the global variable, because there is no assignment statement or global statement for it in that function. If you run `sameName3.py`, the output will look like this:

```
spam
```

In a function, a variable will either always be global or always be local. There's no way that the code in a function can use a local variable named `eggs` and then later in that same function use the global `eggs` variable.

NOTE

If you ever want to modify the value stored in a global variable from in a function, you must use a global statement on that variable.

If you try to use a local variable in a function before you assign a value to it, as in the following program, Python will give you an error. To see this, type the following into the file editor and save it as `sameName4.py`:

```
def spam():
    print(eggs) # ERROR!
❶    eggs = 'spam local'

❷ eggs = 'global'
spam()
```

If you run the previous program, it produces an error message.

```
Traceback (most recent call last):
  File "C:/test3784.py", line 6, in <module>
    spam()
  File "C:/test3784.py", line 2, in spam
    print(eggs) # ERROR!
UnboundLocalError: local variable 'eggs' referenced before assignment
```

This error happens because Python sees that there is an assignment statement for `eggs` in the `spam()` function ❶ and therefore considers `eggs` to be local. But because `print(eggs)` is executed before `eggs` is assigned anything, the local variable `eggs` doesn't exist. Python will *not* fall back to using the global `eggs` variable ❷.

FUNCTIONS AS "BLACK BOXES"

Often, all you need to know about a function are its inputs (the parameters) and output value; you don't always have to burden yourself with how the function's code actually works. When you think about functions in this high-level way, it's common to say that you're treating the function as a "black box."

This idea is fundamental to modern programming. Later chapters in this book will show you several modules with functions that were written by other people. While you can take a peek at the source code if you're curious, you don't need to know how these functions work in order to use them. And because writing functions without global variables is encouraged, you usually don't have to worry about the function's code interacting with the rest of your program.

Exception Handling

Right now, getting an error, or *exception*, in your Python program means the entire program will crash. You don't want this to happen in real-world programs. Instead, you want the program to detect errors, handle them, and then continue to run.

For example, consider the following program, which has a "divide-by-zero" error. Open a new file editor window and enter the following code, saving it as *zeroDivide.py*:

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

We've defined a function called `spam`, given it a parameter, and then printed the value of that function with various parameters to see what happens. This is the output you get when you run the previous code:

```
21.0
3.5
Traceback (most recent call last):
  File "C:/zeroDivide.py", line 6, in <module>
    print(spam(0))
  File "C:/zeroDivide.py", line 2, in spam
    return 42 / divideBy
ZeroDivisionError: division by zero
```

A `ZeroDivisionError` happens whenever you try to divide a number by zero. From the line number given in the error message, you know that the `return` statement in `spam()` is causing an error.

Errors can be handled with try and except statements. The code that could potentially have an error is put in a try clause. The program execution moves to the start of a following except clause if an error happens.

You can put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs.

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

When code in a try clause causes an error, the program execution immediately moves to the code in the except clause. After running that code, the execution continues as normal. The output of the previous program is as follows:

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

Note that any errors that occur in function calls in a try block will also be caught. Consider the following program, which instead has the `spam()` calls in the try block:

```
def spam(divideBy):
    return 42 / divideBy

try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

When this program is run, the output looks like this:

```
21.0
3.5
Error: Invalid argument.
```

The reason `print(spam(1))` is never executed is because once the execution jumps to the code in the `except` clause, it does not return to the `try` clause. Instead, it just continues moving down as normal.

A Short Program: Guess the Number

The toy examples I've shown you so far are useful for introducing basic concepts, but now let's see how everything you've learned comes together in a more complete program. In this section, I'll show you a simple "guess the number" game. When you run this program, the output will look something like this:

```
I am thinking of a number between 1 and 20.  
Take a guess.  
10  
Your guess is too low.  
Take a guess.  
15  
Your guess is too low.  
Take a guess.  
17  
Your guess is too high.  
Take a guess.  
16  
Good job! You guessed my number in 4 guesses!
```

Type the following source code into the file editor, and save the file as `guessTheNumber.py`:

```
# This is a guess the number game.  
import random  
secretNumber = random.randint(1, 20)  
print('I am thinking of a number between 1 and 20.')  
  
# Ask the player to guess 6 times.  
for guessesTaken in range(1, 7):  
    print('Take a guess.')  
    guess = int(input())  
  
    if guess < secretNumber:  
        print('Your guess is too low.')  
    elif guess > secretNumber:  
        print('Your guess is too high.')  
    else:  
        break      # This condition is the correct guess!  
  
if guess == secretNumber:  
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')  
else:  
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

Let's look at this code line by line, starting at the top.

```
# This is a guess the number game.  
import random  
secretNumber = random.randint(1, 20)
```

First, a comment at the top of the code explains what the program does. Then, the program imports the `random` module so that it can use the `random.randint()` function to generate a number for the user to guess. The return value, a random integer between 1 and 20, is stored in the variable `secretNumber`.

```
print('I am thinking of a number between 1 and 20.')  
  
# Ask the player to guess 6 times.  
for guessesTaken in range(1, 7):  
    print('Take a guess.')  
    guess = int(input())
```

The program tells the player that it has come up with a secret number and will give the player six chances to guess it. The code that lets the player enter a guess and checks that guess is in a `for` loop that will loop at most six times. The first thing that happens in the loop is that the player types in a guess. Since `input()` returns a string, its return value is passed straight into `int()`, which translates the string into an integer value. This gets stored in a variable named `guess`.

```
if guess < secretNumber:  
    print('Your guess is too low.')  
elif guess > secretNumber:  
    print('Your guess is too high.)
```

These few lines of code check to see whether the guess is less than or greater than the secret number. In either case, a hint is printed to the screen.

```
else:  
    break # This condition is the correct guess!
```

If the guess is neither higher nor lower than the secret number, then it must be equal to the secret number, in which case you want the program execution to break out of the `for` loop.

```
if guess == secretNumber:  
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')  
else:  
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

After the `for` loop, the previous `if...else` statement checks whether the player has correctly guessed the number and prints an appropriate message to the screen. In both cases, the program displays a variable that contains

an integer value (`guessesTaken` and `secretNumber`). Since it must concatenate these integer values to strings, it passes these variables to the `str()` function, which returns the string value form of these integers. Now these strings can be concatenated with the `+` operators before finally being passed to the `print()` function call.

Summary

Functions are the primary way to compartmentalize your code into logical groups. Since the variables in functions exist in their own local scopes, the code in one function cannot directly affect the values of variables in other functions. This limits what code could be changing the values of your variables, which can be helpful when it comes to debugging your code.

Functions are a great tool to help you organize your code. You can think of them as black boxes: They have inputs in the form of parameters and outputs in the form of return values, and the code in them doesn't affect variables in other functions.

In previous chapters, a single error could cause your programs to crash. In this chapter, you learned about `try` and `except` statements, which can run code when an error has been detected. This can make your programs more resilient to common error cases.

Practice Questions

1. Why are functions advantageous to have in your programs?
2. When does the code in a function execute: when the function is defined or when the function is called?
3. What statement creates a function?
4. What is the difference between a function and a function call?
5. How many global scopes are there in a Python program? How many local scopes?
6. What happens to variables in a local scope when the function call returns?
7. What is a return value? Can a return value be part of an expression?
8. If a function does not have a return statement, what is the return value of a call to that function?
9. How can you force a variable in a function to refer to the global variable?
10. What is the data type of `None`?
11. What does the `import areallyourpetsnamederic` statement do?
12. If you had a function named `bacon()` in a module named `spam`, how would you call it after importing `spam`?
13. How can you prevent a program from crashing when it gets an error?
14. What goes in the `try` clause? What goes in the `except` clause?

Practice Projects

For practice, write programs to do the following tasks.

The Collatz Sequence

Write a function named `collatz()` that has one parameter named `number`. If `number` is even, then `collatz()` should print `number // 2` and return this value. If `number` is odd, then `collatz()` should print and return `3 * number + 1`.

Then write a program that lets the user type in an integer and that keeps calling `collatz()` on that number until the function returns the value 1. (Amazingly enough, this sequence actually works for any integer—sooner or later, using this sequence, you'll arrive at 1! Even mathematicians aren't sure why. Your program is exploring what's called the *Collatz sequence*, sometimes called "the simplest impossible math problem.")

Remember to convert the return value from `input()` to an integer with the `int()` function; otherwise, it will be a string value.

Hint: An integer `number` is even if `number % 2 == 0`, and it's odd if `number % 2 == 1`.

The output of this program could look something like this:

Enter number:

3
10
5
16
8
4
2
1

Input Validation

Add `try` and `except` statements to the previous project to detect whether the user types in a noninteger string. Normally, the `int()` function will raise a `ValueError` error if it is passed a noninteger string, as in `int('puppy')`. In the `except` clause, print a message to the user saying they must enter an integer.

4

LISTS



One more topic you'll need to understand before you can begin writing programs in earnest is the list data type and its cousin, the tuple. Lists and tuples can contain multiple values, which makes it easier to write programs that handle large amounts of data. And since lists themselves can contain other lists, you can use them to arrange data into hierarchical structures.

In this chapter, I'll discuss the basics of lists. I'll also teach you about methods, which are functions that are tied to values of a certain data type. Then I'll briefly cover the list-like tuple and string data types and how they compare to list values. In the next chapter, I'll introduce you to the dictionary data type.

The List Data Type

A *list* is a value that contains multiple values in an ordered sequence. The term *list value* refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value), not the values inside the list value. A list value looks like this: `['cat', 'bat', 'rat', 'elephant']`. Just as string values are typed with quote characters to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, `[]`. Values inside the list are also called *items*. Items are separated with commas (that is, they are *comma-delimited*). For example, enter the following into the interactive shell:

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ['cat', 'bat', 'rat', 'elephant']
['cat', 'bat', 'rat', 'elephant']
>>> ['hello', 3.1415, True, None, 42]
['hello', 3.1415, True, None, 42]
❶ >>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

The `spam` variable ❶ is still assigned only one value: the list value. But the list value itself contains other values. The value `[]` is an empty list that contains no values, similar to `''`, the empty string.

Getting Individual Values in a List with Indexes

Say you have the list `['cat', 'bat', 'rat', 'elephant']` stored in a variable named `spam`. The Python code `spam[0]` would evaluate to `'cat'`, and `spam[1]` would evaluate to `'bat'`, and so on. The integer inside the square brackets that follows the list is called an *index*. The first value in the list is at index `0`, the second value is at index `1`, the third value is at index `2`, and so on. Figure 4-1 shows a list value assigned to `spam`, along with what the index expressions would evaluate to.

For example, type the following expressions into the interactive shell. Start by assigning a list to the variable `spam`.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
>>> spam[1]
'bat'
>>> spam[2]
'rat'
>>> spam[3]
'elephant'
```

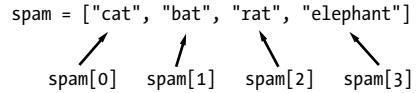


Figure 4-1: A list value stored in the variable `spam`, showing which value each index refers to

```
>>> ['cat', 'bat', 'rat', 'elephant'][3]
'elephant'
❶ >>> 'Hello ' + spam[0]
❷ 'Hello cat'
>>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
'The bat ate the cat.'
```

Notice that the expression 'Hello ' + spam[0] ❶ evaluates to 'Hello ' + 'cat' because spam[0] evaluates to the string 'cat'. This expression in turn evaluates to the string value 'Hello cat' ❷.

Python will give you an `IndexError` error message if you use an index that exceeds the number of values in your list value.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[10000]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    spam[10000]
IndexError: list index out of range
```

Indexes can be only integer values, not floats. The following example will cause a `TypeError` error:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1]
'bat'
>>> spam[1.0]
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    spam[1.0]
TypeError: list indices must be integers, not float
>>> spam[int(1.0)]
'bat'
```

Lists can also contain other list values. The values in these lists of lists can be accessed using multiple indexes, like so:

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

The first index dictates which list value to use, and the second indicates the value within the list value. For example, `spam[0][1]` prints 'bat', the second value in the first list. If you only use one index, the program will print the full list value at that index.

Negative Indexes

While indexes start at 0 and go up, you can also use negative integers for the index. The integer value -1 refers to the last index in a list, the value -2 refers to the second-to-last index in a list, and so on. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]
'elephant'
>>> spam[-3]
'bat'
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
'The elephant is afraid of the bat.'
```

Getting Sublists with Slices

Just as an index can get a single value from a list, a *slice* can get several values from a list, in the form of a new list. A slice is typed between square brackets, like an index, but it has two integers separated by a colon. Notice the difference between indexes and slices.

- `spam[2]` is a list with an index (one integer).
- `spam[1:4]` is a list with a slice (two integers).

In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends. A slice goes up to, but will not include, the value at the second index. A slice evaluates to a new list value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
['bat', 'rat']
>>> spam[0:-1]
['cat', 'bat', 'rat']
```

As a shortcut, you can leave out one or both of the indexes on either side of the colon in the slice. Leaving out the first index is the same as using 0, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
```

```
>>> spam[:]  
['cat', 'bat', 'rat', 'elephant']
```

Getting a List's Length with len()

The `len()` function will return the number of values that are in a list value passed to it, just like it can count the number of characters in a string value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']  
>>> len(spam)  
3
```

Changing Values in a List with Indexes

Normally a variable name goes on the left side of an assignment statement, like `spam = 42`. However, you can also use an index of a list to change the value at that index. For example, `spam[1] = 'aardvark'` means “Assign the value at index 1 in the list `spam` to the string ‘aardvark’.” Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']  
>>> spam[1] = 'aardvark'  
>>> spam  
['cat', 'aardvark', 'rat', 'elephant']  
>>> spam[2] = spam[1]  
>>> spam  
['cat', 'aardvark', 'aardvark', 'elephant']  
>>> spam[-1] = 12345  
>>> spam  
['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

The `+` operator can combine two lists to create a new list value in the same way it combines two strings into a new string value. The `*` operator can also be used with a list and an integer value to replicate the list. Enter the following into the interactive shell:

```
>>> [1, 2, 3] + ['A', 'B', 'C']  
[1, 2, 3, 'A', 'B', 'C']  
>>> ['X', 'Y', 'Z'] * 3  
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']  
>>> spam = [1, 2, 3]  
>>> spam = spam + ['A', 'B', 'C']  
>>> spam  
[1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with `del` Statements

The `del` statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index. For example, enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat']
```

The `del` statement can also be used on a simple variable to delete it, as if it were an “unassignment” statement. If you try to use the variable after deleting it, you will get a `NameError` error because the variable no longer exists.

In practice, you almost never need to delete simple variables. The `del` statement is mostly used to delete values from lists.

Working with Lists

When you first begin writing programs, it’s tempting to create many individual variables to store a group of similar values. For example, if I wanted to store the names of my cats, I might be tempted to write code like this:

```
catName1 = 'Zophie'
catName2 = 'Pooka'
catName3 = 'Simon'
catName4 = 'Lady Macbeth'
catName5 = 'Fat-tail'
catName6 = 'Miss Cleo'
```

(I don’t actually own this many cats, I swear.) It turns out that this is a bad way to write code. For one thing, if the number of cats changes, your program will never be able to store more cats than you have variables. These types of programs also have a lot of duplicate or nearly identical code in them. Consider how much duplicate code is in the following program, which you should enter into the file editor and save as `allMyCats1.py`:

```
print('Enter the name of cat 1:')
catName1 = input()
print('Enter the name of cat 2:')
catName2 = input()
print('Enter the name of cat 3:')
catName3 = input()
print('Enter the name of cat 4:')
catName4 = input()
print('Enter the name of cat 5:')
catName5 = input()
```

```
print('Enter the name of cat 6:')
catName6 = input()
print('The cat names are:')
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' +
catName5 + ' ' + catName6)
```

Instead of using multiple, repetitive variables, you can use a single variable that contains a list value. For example, here's a new and improved version of the *allMyCats1.py* program. This new version uses a single list and can store any number of cats that the user types in. In a new file editor window, type the following source code and save it as *allMyCats2.py*:

```
catNames = []
while True:
    print('Enter the name of cat ' + str(len(catNames) + 1) +
          '(Or enter nothing to stop.)')
    name = input()
    if name == '':
        break
    catNames = catNames + [name] # list concatenation
print('The cat names are:')
for name in catNames:
    print(' ' + name)
```

When you run this program, the output will look something like this:

```
Enter the name of cat 1 (Or enter nothing to stop.):
Zophie
Enter the name of cat 2 (Or enter nothing to stop.):
Pooka
Enter the name of cat 3 (Or enter nothing to stop.):
Simon
Enter the name of cat 4 (Or enter nothing to stop.):
Lady Macbeth
Enter the name of cat 5 (Or enter nothing to stop.):
Fat-tail
Enter the name of cat 6 (Or enter nothing to stop.):
Miss Cleo
Enter the name of cat 7 (Or enter nothing to stop.):

The cat names are:
Zophie
Pooka
Simon
Lady Macbeth
Fat-tail
Miss Cleo
```

The benefit of using a list is that your data is now in a structure, so your program is much more flexible in processing the data than it would be with several repetitive variables.

Using for Loops with Lists

In Chapter 2, you learned about using for loops to execute a block of code a certain number of times. Technically, a for loop repeats the code block once for each value in a list or list-like value. For example, if you ran this code:

```
for i in range(4):
    print(i)
```

the output of this program would be as follows:

```
0
1
2
3
```

This is because the return value from `range(4)` is a list-like value that Python considers similar to `[0, 1, 2, 3]`. The following program has the same output as the previous one:

```
for i in [0, 1, 2, 3]:
    print(i)
```

What the previous for loop actually does is loop through its clause with the variable `i` set to a successive value in the `[0, 1, 2, 3]` list in each iteration.

NOTE

In this book, I use the term list-like to refer to data types that are technically named sequences. You don't need to know the technical definitions of this term, though.

A common Python technique is to use `range(len(someList))` with a for loop to iterate over the indexes of a list. For example, enter the following into the interactive shell:

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
>>> for i in range(len(supplies)):
    print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
```

```
Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flame-throwers
Index 3 in supplies is: binders
```

Using `range(len(supplies))` in the previously shown for loop is handy because the code in the loop can access the index (as the variable `i`) and the value at that index (as `supplies[i]`). Best of all, `range(len(supplies))` will iterate through all the indexes of `supplies`, no matter how many items it contains.

The `in` and `not in` Operators

You can determine whether a value is or isn't in a list with the `in` and `not in` operators. Like other operators, `in` and `not in` are used in expressions and connect two values: a value to look for in a list and the list where it may be found. These expressions will evaluate to a Boolean value. Enter the following into the interactive shell:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam
True
```

For example, the following program lets the user type in a pet name and then checks to see whether the name is in a list of pets. Open a new file editor window, enter the following code, and save it as *myPets.py*:

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
if name not in myPets:
    print('I do not have a pet named ' + name)
else:
    print(name + ' is my pet.')
```

The output may look something like this:

```
Enter a pet name:
Footfoot
I do not have a pet named Footfoot
```

The *Multiple Assignment Trick*

The *multiple assignment trick* is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
>>> cat = ['fat', 'black', 'loud']
>>> size = cat[0]
>>> color = cat[1]
>>> disposition = cat[2]
```

you could type this line of code:

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition = cat
```

The number of variables and the length of the list must be exactly equal, or Python will give you a `ValueError`:

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition, name = cat
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    size, color, disposition, name = cat
ValueError: need more than 3 values to unpack
```

Augmented Assignment Operators

When assigning a value to a variable, you will frequently use the variable itself. For example, after assigning 42 to the variable `spam`, you would increase the value in `spam` by 1 with the following code:

```
>>> spam = 42
>>> spam = spam + 1
>>> spam
43
```

As a shortcut, you can use the augmented assignment operator `+=` to do the same thing:

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

There are augmented assignment operators for the `+`, `-`, `*`, `/`, and `%` operators, described in Table 4-1.

Table 4-1: The Augmented Assignment Operators

Augmented assignment statement	Equivalent assignment statement
<code>spam = spam + 1</code>	<code>spam += 1</code>
<code>spam = spam - 1</code>	<code>spam -= 1</code>
<code>spam = spam * 1</code>	<code>spam *= 1</code>
<code>spam = spam / 1</code>	<code>spam /= 1</code>
<code>spam = spam % 1</code>	<code>spam %= 1</code>

The `+=` operator can also do string and list concatenation, and the `*=` operator can do string and list replication. Enter the following into the interactive shell:

```
>>> spam = 'Hello'
>>> spam += ' world!'
>>> spam
'Hello world!'
```

```
>>> bacon = ['Zophie']
>>> bacon *= 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

Methods

A *method* is the same thing as a function, except it is “called on” a value. For example, if a list value were stored in `spam`, you would call the `index()` list method (which I’ll explain next) on that list like so: `spam.index('hello')`. The method part comes after the value, separated by a period.

Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list.

Finding a Value in a List with the `index()` Method

List values have an `index()` method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value isn’t in the list, then Python produces a `ValueError` error. Enter the following into the interactive shell:

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
>>> spam.index('howdy howdy howdy')
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    spam.index('howdy howdy howdy')
ValueError: 'howdy howdy howdy' is not in list
```

When there are duplicates of the value in the list, the index of its first appearance is returned. Enter the following into the interactive shell, and notice that `index()` returns 1, not 3:

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
>>> spam.index('Pooka')
1
```

Adding Values to Lists with the `append()` and `insert()` Methods

To add new values to a list, use the `append()` and `insert()` methods. Enter the following into the interactive shell to call the `append()` method on a list value stored in the variable `spam`:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
```

```
>>> spam  
['cat', 'dog', 'bat', 'moose']
```

The previous `append()` method call adds the argument to the end of the list. The `insert()` method can insert a value at any index in the list. The first argument to `insert()` is the index for the new value, and the second argument is the new value to be inserted. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'bat']  
>>> spam.insert(1, 'chicken')  
>>> spam  
['cat', 'chicken', 'dog', 'bat']
```

Notice that the code is `spam.append('moose')` and `spam.insert(1, 'chicken')`, not `spam = spam.append('moose')` and `spam = spam.insert(1, 'chicken')`. Neither `append()` nor `insert()` gives the new value of `spam` as its return value. (In fact, the return value of `append()` and `insert()` is `None`, so you definitely wouldn't want to store this as the new variable value.) Rather, the list is modified *in place*. Modifying a list in place is covered in more detail later in “Mutable and Immutable Data Types” on page 94.

Methods belong to a single data type. The `append()` and `insert()` methods are list methods and can be called only on list values, not on other values such as strings or integers. Enter the following into the interactive shell, and note the `AttributeError` error messages that show up:

```
>>> eggs = 'hello'  
>>> eggs.append('world')  
Traceback (most recent call last):  
  File "<pyshell#19>", line 1, in <module>  
    eggs.append('world')  
AttributeError: 'str' object has no attribute 'append'  
>>> bacon = 42  
>>> bacon.insert(1, 'world')  
Traceback (most recent call last):  
  File "<pyshell#22>", line 1, in <module>  
    bacon.insert(1, 'world')  
AttributeError: 'int' object has no attribute 'insert'
```

Removing Values from Lists with `remove()`

The `remove()` method is passed the value to be removed from the list it is called on. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']  
>>> spam.remove('bat')  
>>> spam  
['cat', 'rat', 'elephant']
```

Attempting to delete a value that does not exist in the list will result in a `ValueError` error. For example, enter the following into the interactive shell and notice the error that is displayed:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('chicken')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    spam.remove('chicken')
ValueError: list.remove(x): x not in list
```

If the value appears multiple times in the list, only the first instance of the value will be removed. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']
>>> spam.remove('cat')
>>> spam
['bat', 'rat', 'cat', 'hat', 'cat']
```

The `del` statement is good to use when you know the index of the value you want to remove from the list. The `remove()` method is good when you know the value you want to remove from the list.

Sorting the Values in a List with the `sort()` Method

Lists of number values or lists of strings can be sorted with the `sort()` method. For example, enter the following into the interactive shell:

```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

You can also pass `True` for the `reverse` keyword argument to have `sort()` sort the values in reverse order. Enter the following into the interactive shell:

```
>>> spam.sort(reverse=True)
>>> spam
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

There are three things you should note about the `sort()` method. First, the `sort()` method sorts the list in place; don't try to capture the return value by writing code like `spam = spam.sort()`.

Second, you cannot sort lists that have both number values *and* string values in them, since Python doesn't know how to compare these values. Type the following into the interactive shell and notice the `TypeError` error:

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    spam.sort()
TypeError: unorderable types: str() < int()
```

Third, `sort()` uses “ASCIIbetical order” rather than actual alphabetical order for sorting strings. This means uppercase letters come before lowercase letters. Therefore, the lowercase *a* is sorted so that it comes *after* the uppercase *Z*. For an example, enter the following into the interactive shell:

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
>>> spam.sort()
>>> spam
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

If you need to sort the values in regular alphabetical order, pass `str.lower` for the `key` keyword argument in the `sort()` method call.

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

This causes the `sort()` function to treat all the items in the list as if they were lowercase without actually changing the values in the list.

Example Program: Magic 8 Ball with a List

Using lists, you can write a much more elegant version of the previous chapter’s Magic 8 Ball program. Instead of several lines of nearly identical `elif` statements, you can create a single list that the code works with. Open a new file editor window and enter the following code. Save it as *magic8Ball2.py*.

```
import random

messages = ['It is certain',
            'It is decidedly so',
            'Yes definitely',
            'Reply hazy try again',
            'Ask again later',
            'Concentrate and ask again',
            'My reply is no',
            'Outlook not so good',
            'Very doubtful']

print(messages[random.randint(0, len(messages) - 1)])
```

EXCEPTIONS TO INDENTATION RULES IN PYTHON

In most cases, the amount of indentation for a line of code tells Python what block it is in. There are some exceptions to this rule, however. For example, lists can actually span several lines in the source code file. The indentation of these lines do not matter; Python knows that until it sees the ending square bracket, the list is not finished. For example, you can have code that looks like this:

```
spam = ['apples',
        'oranges',
        'bananas',
        'cats']
print(spam)
```

Of course, practically speaking, most people use Python's behavior to make their lists look pretty and readable, like the `messages` list in the `Magic 8 Ball` program.

You can also split up a single instruction across multiple lines using the \ line continuation character at the end. Think of \ as saying, "This instruction continues on the next line." The indentation on the line after a \ line continuation is not significant. For example, the following is valid Python code:

```
print('Four score and seven ' +
      'years ago...')
```

These tricks are useful when you want to rearrange long lines of Python code to be a bit more readable.

When you run this program, you'll see that it works the same as the previous `magic8Ball.py` program.

Notice the expression you use as the index into `messages: random.randint(0, len(messages) - 1)`. This produces a random number to use for the index, regardless of the size of `messages`. That is, you'll get a random number between 0 and the value of `len(messages) - 1`. The benefit of this approach is that you can easily add and remove strings to the `messages` list without changing other lines of code. If you later update your code, there will be fewer lines you have to change and fewer chances for you to introduce bugs.

List-like Types: Strings and Tuples

Lists aren't the only data types that represent ordered sequences of values. For example, strings and lists are actually similar, if you consider a string to be a "list" of single text characters. Many of the things you can do with lists

can also be done with strings: indexing; slicing; and using them with for loops, with len(), and with the in and not in operators. To see this, enter the following into the interactive shell:

```
>>> name = 'Zophie'
>>> name[0]
'Z'
>>> name[-2]
'i'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'z' in name
False
>>> 'p' not in name
False
>>> for i in name:
    print('*****' + i + '*****')

***** Z *****
***** o *****
***** p *****
***** h *****
***** i *****
***** e *****
```

Mutable and Immutable Data Types

But lists and strings are different in an important way. A list value is a *mutable* data type: It can have values added, removed, or changed. However, a string is *immutable*: It cannot be changed. Trying to reassign a single character in a string results in a `TypeError` error, as you can see by entering the following into the interactive shell:

```
>>> name = 'Zophie a cat'
>>> name[7] = 'the'
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    name[7] = 'the'
TypeError: 'str' object does not support item assignment
```

The proper way to “mutate” a string is to use slicing and concatenation to build a *new* string by copying from parts of the old string. Enter the following into the interactive shell:

```
>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
>>> name
'Zophie a cat'
```

```
>>> newName  
'Zophie the cat'
```

We used [0:7] and [8:12] to refer to the characters that we don't wish to replace. Notice that the original 'Zophie a cat' string is not modified because strings are immutable.

Although a list value *is* mutable, the second line in the following code does not modify the list eggs:

```
>>> eggs = [1, 2, 3]  
>>> eggs = [4, 5, 6]  
>>> eggs  
[4, 5, 6]
```

The list value in eggs isn't being changed here; rather, an entirely new and different list value ([4, 5, 6]) is overwriting the old list value ([1, 2, 3]). This is depicted in Figure 4-2.

If you wanted to actually modify the original list in eggs to contain [4, 5, 6], you would have to do something like this:

```
>>> eggs = [1, 2, 3]  
>>> del eggs[2]  
>>> del eggs[1]  
>>> del eggs[0]  
>>> eggs.append(4)  
>>> eggs.append(5)  
>>> eggs.append(6)  
>>> eggs  
[4, 5, 6]
```



Figure 4-2: When eggs = [4, 5, 6] is executed, the contents of eggs are replaced with a new list value.

In the first example, the list value that eggs ends up with is the same list value it started with. It's just that this list has been changed, rather than overwritten. Figure 4-3 depicts the seven changes made by the first seven lines in the previous interactive shell example.



Figure 4-3: The `del` statement and the `append()` method modify the same list value in place.

Changing a value of a mutable data type (like what the `del` statement and `append()` method do in the previous example) changes the value in place, since the variable's value is not replaced with a new list value.

Mutable versus immutable types may seem like a meaningless distinction, but “Passing References” on page 100 will explain the different behavior when calling functions with mutable arguments versus immutable arguments. But first, let’s find out about the tuple data type, which is an immutable form of the list data type.

The Tuple Data Type

The *tuple* data type is almost identical to the list data type, except in two ways. First, tuples are typed with parentheses, (and), instead of square brackets, [and]. For example, enter the following into the interactive shell:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1:3]
(42, 0.5)
>>> len(eggs)
3
```

But the main way that tuples are different from lists is that tuples, like strings, are immutable. Tuples cannot have their values modified, appended, or removed. Enter the following into the interactive shell, and look at the `TypeError` error message:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

If you have only one value in your tuple, you can indicate this by placing a trailing comma after the value inside the parentheses. Otherwise, Python will think you've just typed a value inside regular parentheses. The comma is what lets Python know this is a tuple value. (Unlike some other programming languages, in Python it's fine to have a trailing comma after the last item in a list or tuple.) Enter the following `type()` function calls into the interactive shell to see the distinction:

```
>>> type('hello',())
<class 'tuple'>
>>> type('hello')
<class 'str'>
```

You can use tuples to convey to anyone reading your code that you don't intend for that sequence of values to change. If you need an ordered sequence of values that never changes, use a tuple. A second benefit of using tuples instead of lists is that, because they are immutable and their contents don't change, Python can implement some optimizations that make code using tuples slightly faster than code using lists.

Converting Types with the `list()` and `tuple()` Functions

Just like how `str(42)` will return `'42'`, the string representation of the integer 42, the functions `list()` and `tuple()` will return list and tuple versions of the values passed to them. Enter the following into the interactive shell, and notice that the return value is of a different data type than the value passed:

```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
>>> list(('cat', 'dog', 5))
['cat', 'dog', 5]
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

Converting a tuple to a list is handy if you need a mutable version of a tuple value.

References

As you've seen, variables store strings and integer values. Enter the following into the interactive shell:

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

You assign 42 to the `spam` variable, and then you copy the value in `spam` and assign it to the variable `cheese`. When you later change the value in `spam` to 100, this doesn't affect the value in `cheese`. This is because `spam` and `cheese` are different variables that store different values.

But lists don't work this way. When you assign a list to a variable, you are actually assigning a list *reference* to the variable. A reference is a value that points to some bit of data, and a list reference is a value that points to a list. Here is some code that will make this distinction easier to understand. Enter this into the interactive shell:

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam
❸ >>> cheese[1] = 'Hello!'
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

This might look odd to you. The code changed only the `cheese` list, but it seems that both the `cheese` and `spam` lists have changed.

When you create the list ❶, you assign a reference to it in the `spam` variable. But the next line ❷ copies only the list reference in `spam` to `cheese`, not the list value itself. This means the values stored in `spam` and `cheese` now both refer to the same list. There is only one underlying list because the list itself was never actually copied. So when you modify the first element of `cheese` ❸, you are modifying the same list that `spam` refers to.

Remember that variables are like boxes that contain values. The previous figures in this chapter show that lists in boxes aren't exactly accurate because list variables don't actually contain lists—they contain *references* to lists. (These references will have ID numbers that Python uses internally, but you can ignore them.) Using boxes as a metaphor for variables, Figure 4-4 shows what happens when a list is assigned to the `spam` variable.

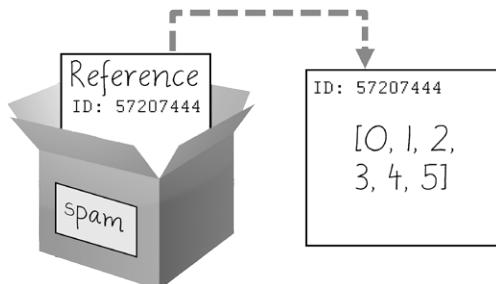


Figure 4-4: `spam = [0, 1, 2, 3, 4, 5]` stores a reference to a list, not the actual list.

Then, in Figure 4-5, the reference in `spam` is copied to `cheese`. Only a new reference was created and stored in `cheese`, not a new list. Note how both references refer to the same list.



Figure 4-5: `spam` = `cheese` copies the reference, not the list.

When you alter the list that `cheese` refers to, the list that `spam` refers to is also changed, because both `cheese` and `spam` refer to the same list. You can see this in Figure 4-6.



Figure 4-6: `cheese[1] = 'Hello!'` modifies the list that both variables refer to.

Variables will contain references to list values rather than list values themselves. But for strings and integer values, variables simply contain the string or integer value. Python uses references whenever variables must store values of mutable data types, such as lists or dictionaries. For values of immutable data types such as strings, integers, or tuples, Python variables will store the value itself.

Although Python variables technically contain references to list or dictionary values, people often casually say that the variable contains the list or dictionary.

Passing References

References are particularly important for understanding how arguments get passed to functions. When a function is called, the values of the arguments are copied to the parameter variables. For lists (and dictionaries, which I'll describe in the next chapter), this means a copy of the reference is used for the parameter. To see the consequences of this, open a new file editor window, enter the following code, and save it as *passingReference.py*:

```
def eggs(someParameter):
    someParameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)
```

Notice that when `eggs()` is called, a return value is not used to assign a new value to `spam`. Instead, it modifies the list in place, directly. When run, this program produces the following output:

```
[1, 2, 3, 'Hello']
```

Even though `spam` and `someParameter` contain separate references, they both refer to the same list. This is why the `append('Hello')` method call inside the function affects the list even after the function call has returned.

Keep this behavior in mind: Forgetting that Python handles list and dictionary variables this way can lead to confusing bugs.

The `copy` Module's `copy()` and `deepcopy()` Functions

Although passing around references is often the handiest way to deal with lists and dictionaries, if the function modifies the list or dictionary that is passed, you may not want these changes in the original list or dictionary value. For this, Python provides a module named `copy` that provides both the `copy()` and `deepcopy()` functions. The first of these, `copy.copy()`, can be used to make a duplicate copy of a mutable value like a list or dictionary, not just a copy of a reference. Enter the following into the interactive shell:

```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
>>> cheese = copy.copy(spam)
>>> cheese[1] = 42
>>> spam
['A', 'B', 'C', 'D']
>>> cheese
['A', 42, 'C', 'D']
```

Now the `spam` and `cheese` variables refer to separate lists, which is why only the list in `cheese` is modified when you assign 42 at index 7. As you can see in Figure 4-7, the reference ID numbers are no longer the same for both variables because the variables refer to independent lists.



Figure 4-7: `cheese = copy.copy(spam)` creates a second list that can be modified independently of the first.

If the list you need to copy contains lists, then use the `copy.deepcopy()` function instead of `copy.copy()`. The `deepcopy()` function will copy these inner lists as well.

Summary

Lists are useful data types since they allow you to write code that works on a modifiable number of values in a single variable. Later in this book, you will see programs using lists to do things that would be difficult or impossible to do without them.

Lists are mutable, meaning that their contents can change. Tuples and strings, although list-like in some respects, are immutable and cannot be changed. A variable that contains a tuple or string value can be overwritten with a new tuple or string value, but this is not the same thing as modifying the existing value in place—like, say, the `append()` or `remove()` methods do on lists.

Variables do not store list values directly; they store *references* to lists. This is an important distinction when copying variables or passing lists as arguments in function calls. Because the value that is being copied is the list reference, be aware that any changes you make to the list might impact another variable in your program. You can use `copy()` or `deepcopy()` if you want to make changes to a list in one variable without modifying the original list.

Practice Questions

1. What is []?
2. How would you assign the value 'hello' as the third value in a list stored in a variable named `spam`? (Assume `spam` contains [2, 4, 6, 8, 10].)

For the following three questions, let's say `spam` contains the list ['a', 'b', 'c', 'd'].

3. What does `spam[int('3' * 2) / 11]` evaluate to?
4. What does `spam[-1]` evaluate to?
5. What does `spam[:2]` evaluate to?

For the following three questions, let's say `bacon` contains the list [3.14, 'cat', 11, 'cat', True].

6. What does `bacon.index('cat')` evaluate to?
7. What does `bacon.append(99)` make the list value in `bacon` look like?
8. What does `bacon.remove('cat')` make the list value in `bacon` look like?
9. What are the operators for list concatenation and list replication?
10. What is the difference between the `append()` and `insert()` list methods?
11. What are two ways to remove values from a list?
12. Name a few ways that list values are similar to string values.
13. What is the difference between lists and tuples?
14. How do you type the tuple value that has just the integer value 42 in it?
15. How can you get the tuple form of a list value? How can you get the list form of a tuple value?
16. Variables that "contain" list values don't actually contain lists directly. What do they contain instead?
17. What is the difference between `copy.copy()` and `copy.deepcopy()`?

Practice Projects

For practice, write programs to do the following tasks.

Comma Code

Say you have a list value like this:

```
spam = ['apples', 'bananas', 'tofu', 'cats']
```

Write a function that takes a list value as an argument and returns a string with all the items separated by a comma and a space, with *and* inserted before the last item. For example, passing the previous `spam` list to the function would return 'apples, bananas, tofu, and cats'. But your function should be able to work with any list value passed to it.

Character Picture Grid

Say you have a list of lists where each value in the inner lists is a one-character string, like this:

```
grid = [['.', '.', '.', '.', '.', '.', '.'],
        ['.', '0', '0', '.', '.', '.', '.'],
        ['0', '0', '0', '0', '.', '.', '.'],
        ['0', '0', '0', '0', '0', '.', '.'],
        ['.', '0', '0', '0', '0', '0', '0'],
        ['0', '0', '0', '0', '0', '0', '0'],
        ['0', '0', '0', '0', '0', '0', '.'],
        ['0', '0', '0', '0', '.', '.', '.'],
        ['.', '0', '0', '.', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.', '.']]
```

You can think of `grid[x][y]` as being the character at the x- and y-coordinates of a “picture” drawn with text characters. The (0, 0) origin will be in the upper-left corner, the x-coordinates increase going right, and w the y-coordinates increase going down.

Copy the previous grid value, and write code that uses it to print the image.

```
.00.00..
.000000.
.000000.
..0000..
...00...
....0....
```

Hint: You will need to use a loop in a loop in order to print `grid[0][0]`, then `grid[1][0]`, then `grid[2][0]`, and so on, up to `grid[8][0]`. This will finish the first row, so then print a newline. Then your program should print `grid[0][1]`, then `grid[1][1]`, then `grid[2][1]`, and so on. The last thing your program will print is `grid[8][5]`.

Also, remember to pass the `end` keyword argument to `print()` if you don’t want a newline printed automatically after each `print()` call.

5

DICTIONARIES AND STRUCTURING DATA



In this chapter, I will cover the dictionary data type, which provides a flexible way to access and organize data. Then, combining dictionaries with your knowledge of lists from the previous chapter, you'll learn how to create a data structure to model a tic-tac-toe board.

The Dictionary Data Type

Like a list, a *dictionary* is a collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called *keys*, and a key with its associated value is called a *key-value pair*.

In code, a dictionary is typed with braces, `{}`. Enter the following into the interactive shell:

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

This assigns a dictionary to the `myCat` variable. This dictionary's keys are `'size'`, `'color'`, and `'disposition'`. The values for these keys are `'fat'`, `'gray'`, and `'loud'`, respectively. You can access these values through their keys:

```
>>> myCat['size']
'fat'
>>> 'My cat has ' + myCat['color'] + ' fur.'
'My cat has gray fur.'
```

Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they do not have to start at `0` and can be any number.

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

Dictionaries vs. Lists

Unlike lists, items in dictionaries are unordered. The first item in a list named `spam` would be `spam[0]`. But there is no “first” item in a dictionary. While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary. Enter the following into the interactive shell:

```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```

Because dictionaries are not ordered, they can't be sliced like lists.

Trying to access a key that does not exist in a dictionary will result in a `KeyError` error message, much like a list's “out-of-range” `IndexError` error message. Enter the following into the interactive shell, and notice the error message that shows up because there is no `'color'` key:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    spam['color']
KeyError: 'color'
```

Though dictionaries are not ordered, the fact that you can have arbitrary values for the keys allows you to organize your data in powerful ways. Say you wanted your program to store data about your friends' birthdays. You can use a dictionary with the names as keys and the birthdays as values. Open a new file editor window and enter the following code. Save it as `birthdays.py`.

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

    while True:
        print('Enter a name: (blank to quit)')
        name = input()
        if name == '':
            break

❷    if name in birthdays:
❸        print(birthdays[name] + ' is the birthday of ' + name)
    else:
        print('I do not have birthday information for ' + name)
        print('What is their birthday?')
        bday = input()
❹    birthdays[name] = bday
    print('Birthday database updated.')
```

You create an initial dictionary and store it in `birthdays` ❶. You can see if the entered name exists as a key in the dictionary with the `in` keyword ❷, just as you did for lists. If the name is in the dictionary, you access the associated value using square brackets ❸; if not, you can add it using the same square bracket syntax combined with the assignment operator ❹.

When you run this program, it will look like this:

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

Of course, all the data you enter in this program is forgotten when the program terminates. You'll learn how to save data to files on the hard drive in Chapter 8.

The `keys()`, `values()`, and `items()` Methods

There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: `keys()`, `values()`, and `items()`. The values returned by these methods are not true lists: They cannot be modified and do not have an `append()` method. But these data types (`dict_keys`,

`dict_values`, and `dict_items`, respectively) *can* be used in for loops. To see how these methods work, enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
    print(v)

red
42
```

Here, a for loop iterates over each of the values in the `spam` dictionary. A for loop can also iterate over the keys or both keys and values:

```
>>> for k in spam.keys():
    print(k)

color
age
>>> for i in spam.items():
    print(i)

('color', 'red')
('age', 42)
```

Using the `keys()`, `values()`, and `items()` methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively. Notice that the values in the `dict_items` value returned by the `items()` method are tuples of the key and value.

If you want a true list from one of these methods, pass its list-like return value to the `list()` function. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

The `list(spam.keys())` line takes the `dict_keys` value returned from `keys()` and passes it to `list()`, which then returns a list value of `['color', 'age']`.

You can also use the multiple assignment trick in a for loop to assign the key and value to separate variables. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
    print('Key: ' + k + ' Value: ' + str(v))

Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

Recall from the previous chapter that the `in` and `not in` operators can check whether a value exists in a list. You can also use these operators to see whether a certain key or value exists in a dictionary. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

In the previous example, notice that `'color' in spam` is essentially a shorter version of writing `'color' in spam.keys()`. This is always the case: If you ever want to check whether a value is (or isn't) a key in the dictionary, you can simply use the `in` (or `not in`) keyword with the dictionary value itself.

The `get()` Method

It's tedious to check whether a key exists in a dictionary before accessing that key's value. Fortunately, dictionaries have a `get()` method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

Enter the following into the interactive shell:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

Because there is no 'eggs' key in the `picnicItems` dictionary, the default value 0 is returned by the `get()` method. Without using `get()`, the code would have caused an error message, such as in the following example:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
KeyError: 'eggs'
```

The `setdefault()` Method

You'll often have to set a value in a dictionary for a certain key only if that key does not already have a value. The code looks something like this:

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

The `setdefault()` method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the `setdefault()` method returns the key's value. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

The first time `setdefault()` is called, the dictionary in `spam` changes to `{'color': 'black', 'age': 5, 'name': 'Pooka'}`. The method returns the value 'black' because this is now the value set for the key 'color'. When `spam.setdefault('color', 'white')` is called next, the value for that key is *not* changed to 'white' because `spam` already has a key named 'color'.

The `setdefault()` method is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a string. Open the file editor window and enter the following code, saving it as `characterCount.py`:

```
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

The program loops over each character in the `message` variable's string, counting how often each character appears. The `setdefault()` method call ensures that the key is in the `count` dictionary (with a default value of 0)

so the program doesn't throw a `KeyError` error when `count[character] = count[character] + 1` is executed. When you run this program, the output will look like this:

```
{' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2, 'i': 6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1}
```

From the output, you can see that the lowercase letter *c* appears 3 times, the space character appears 13 times, and the uppercase letter *A* appears 1 time. This program will work no matter what string is inside the `message` variable, even if the string is millions of characters long!

Pretty Printing

If you import the `pprint` module into your programs, you'll have access to the `pprint()` and `pformat()` functions that will “pretty print” a dictionary's values. This is helpful when you want a cleaner display of the items in a dictionary than what `print()` provides. Modify the previous `characterCount.py` program and save it as `prettyCharacterCount.py`.

```
import pprint
message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

pprint.pprint(count)
```

This time, when the program is run, the output looks much cleaner, with the keys sorted.

```
{' ': 13,
',': 1,
'.': 1,
'A': 1,
'I': 1,
'a': 4,
'b': 1,
'c': 3,
'd': 3,
'e': 5,
'g': 2,
'h': 3,
'i': 6,
```

```
'k': 2,  
'l': 3,  
'n': 4,  
'o': 2,  
'p': 1,  
'r': 5,  
's': 3,  
't': 6,  
'w': 2,  
'y': 1}
```

The `pprint.pprint()` function is especially helpful when the dictionary itself contains nested lists or dictionaries.

If you want to obtain the prettified text as a string value instead of displaying it on the screen, call `pprint.pformat()` instead. These two lines are equivalent to each other:

```
pprint.pprint(someDictionaryValue)  
print(pprint.pformat(someDictionaryValue))
```

Using Data Structures to Model Real-World Things

Even before the Internet, it was possible to play a game of chess with someone on the other side of the world. Each player would set up a chessboard at their home and then take turns mailing a postcard to each other describing each move. To do this, the players needed a way to unambiguously describe the state of the board and their moves.

In *algebraic chess notation*, the spaces on the chessboard are identified by a number and letter coordinate, as in Figure 5-1.



Figure 5-1: The coordinates of a chessboard in algebraic chess notation

The chess pieces are identified by letters: *K* for king, *Q* for queen, *R* for rook, *B* for bishop, and *N* for knight. Describing a move uses the letter of the piece and the coordinates of its destination. A pair of these moves describes

what happens in a single turn (with white going first); for instance, the notation 2. Nf3 Nc6 indicates that white moved a knight to f3 and black moved a knight to c6 on the second turn of the game.

There's a bit more to algebraic notation than this, but the point is that you can use it to unambiguously describe a game of chess without needing to be in front of a chessboard. Your opponent can even be on the other side of the world! In fact, you don't even need a physical chess set if you have a good memory: You can just read the mailed chess moves and update boards you have in your imagination.

Computers have good memories. A program on a modern computer can easily store billions of strings like '2. Nf3 Nc6'. This is how computers can play chess without having a physical chessboard. They model data to represent a chessboard, and you can write code to work with this model.

This is where lists and dictionaries can come in. You can use them to model real-world things, like chessboards. For the first example, you'll use a game that's a little simpler than chess: tic-tac-toe.

A Tic-Tac-Toe Board

A tic-tac-toe board looks like a large hash symbol (#) with nine slots that can each contain an X, an O, or a blank. To represent the board with a dictionary, you can assign each slot a string-value key, as shown in Figure 5-2.

You can use string values to represent what's in each slot on the board: 'X', 'O', or ' ' (a space character). Thus, you'll need to store nine strings. You can use a dictionary of values for this. The string value with the key 'top-R' can represent the top-right corner, the string value with the key 'low-L' can represent the bottom-left corner, the string value with the key 'mid-M' can represent the middle, and so on.

This dictionary is a data structure that represents a tic-tac-toe board. Store this board-as-a-dictionary in a variable named `theBoard`. Open a new file editor window, and enter the following source code, saving it as `ticTacToe.py`:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

The data structure stored in the `theBoard` variable represents the tic-tac-toe board in Figure 5-3.

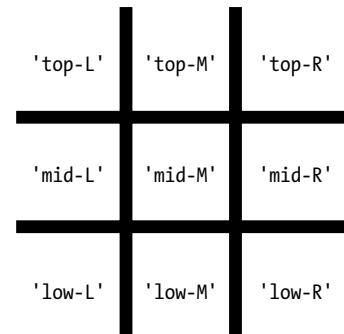


Figure 5-2: The slots of a tic-tac-toe board with their corresponding keys



Figure 5-3: An empty tic-tac-toe board

Since the value for every key in `theBoard` is a single-space string, this dictionary represents a completely clear board. If player X went first and chose the middle space, you could represent that board with this dictionary:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

The data structure in `theBoard` now represents the tic-tac-toe board in Figure 5-4.



Figure 5-4: The first move

A board where player O has won by placing Os across the top might look like this:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
```

The data structure in `theBoard` now represents the tic-tac-toe board in Figure 5-5.



Figure 5-5: Player O wins.

Of course, the player sees only what is printed to the screen, not the contents of variables. Let's create a function to print the board dictionary onto the screen. Make the following addition to *ticTacToe.py* (new code is in bold):

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
           'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
           'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
def printBoard(board):
    print(board['top-L'] + ' | ' + board['top-M'] + ' | ' + board['top-R'])
    print('---+---')
    print(board['mid-L'] + ' | ' + board['mid-M'] + ' | ' + board['mid-R'])
    print('---+---')
    print(board['low-L'] + ' | ' + board['low-M'] + ' | ' + board['low-R'])
printBoard(theBoard)
```

When you run this program, `printBoard()` will print out a blank tic-tac-toe board.

```
| |
---+
| |
---+
| |
```

The `printBoard()` function can handle any tic-tac-toe data structure you pass it. Try changing the code to the following:

```
theBoard = {'top-L': '0', 'top-M': '0', 'top-R': '0', 'mid-L': 'X', 'mid-M':
'X', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
def printBoard(board):
    print(board['top-L'] + ' | ' + board['top-M'] + ' | ' + board['top-R'])
    print('---+---')
    print(board['mid-L'] + ' | ' + board['mid-M'] + ' | ' + board['mid-R'])
    print('---+---')
    print(board['low-L'] + ' | ' + board['low-M'] + ' | ' + board['low-R'])
printBoard(theBoard)
```

Now when you run this program, the new board will be printed to the screen.

```
0|0|0
-+-
X|X|
-+-
| |X
```

Because you created a data structure to represent a tic-tac-toe board and wrote code in `printBoard()` to interpret that data structure, you now have a program that “models” the tic-tac-toe board. You could have organized your data structure differently (for example, using keys like ‘TOP-LEFT’ instead of ‘top-L’), but as long as the code works with your data structures, you will have a correctly working program.

For example, the `printBoard()` function expects the tic-tac-toe data structure to be a dictionary with keys for all nine slots. If the dictionary you passed was missing, say, the ‘mid-L’ key, your program would no longer work.

```
0|0|0
-+-
Traceback (most recent call last):
  File "ticTacToe.py", line 10, in <module>
    printBoard(theBoard)
  File "ticTacToe.py", line 6, in printBoard
    print(board['mid-L'] + ' | ' + board['mid-M'] + ' | ' + board['mid-R'])
KeyError: 'mid-L'
```

Now let’s add code that allows the players to enter their moves. Modify the *ticTacToe.py* program to look like this:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-M': ' ',
           'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
    print(board['top-L'] + ' | ' + board['top-M'] + ' | ' + board['top-R'])
    print('---')
    print(board['mid-L'] + ' | ' + board['mid-M'] + ' | ' + board['mid-R'])
    print('---')
    print(board['low-L'] + ' | ' + board['low-M'] + ' | ' + board['low-R'])
turn = 'X'
for i in range(9):
    ❶    printBoard(theBoard)
    print('Turn for ' + turn + '. Move on which space?')
    ❷    move = input()
    ❸    theBoard[move] = turn
    ❹    if turn == 'X':
        turn = 'O'
    else:
        turn = 'X'
    printBoard(theBoard)
```

The new code prints out the board at the start of each new turn ❶, gets the active player’s move ❷, updates the game board accordingly ❸, and then swaps the active player ❹ before moving on to the next turn.

When you run this program, it will look something like this:

```
| |
-+-
| |
-+-
| |
Turn for X. Move on which space?
mid-M
| |
-+-
|X|
-+-
| |
Turn for O. Move on which space?
low-L
| |
-+-
|X|
-+-
O| |
--snip--

O|O|X
-+-
X|X|O
-+-
O| |X
Turn for X. Move on which space?
low-M
O|O|X
-+-
X|X|O
-+-
O|X|X
```

This isn’t a complete tic-tac-toe game—for instance, it doesn’t ever check whether a player has won—but it’s enough to see how data structures can be used in programs.

NOTE

If you are curious, the source code for a complete tic-tac-toe program is described in the resources available from <http://nostarch.com/automatestuff/>.

Nested Dictionaries and Lists

Modeling a tic-tac-toe board was fairly simple: The board needed only a single dictionary value with nine key-value pairs. As you model more complicated things, you may find you need dictionaries and lists that contain

other dictionaries and lists. Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values. For example, here's a program that uses a dictionary that contains other dictionaries in order to see who is bringing what to a picnic. The `totalBrought()` function can read this data structure and calculate the total number of an item being brought by all the guests.

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
            'Bob': {'ham sandwiches': 3, 'apples': 2},
            'Carol': {'cups': 3, 'apple pies': 1)}

def totalBrought(guests, item):
    numBrought = 0
    ❶   for k, v in guests.items():
        ❷       numBrought = numBrought + v.get(item, 0)
    return numBrought

print('Number of things being brought:')
print(' - Apples      ' + str(totalBrought(allGuests, 'apples')))
print(' - Cups        ' + str(totalBrought(allGuests, 'cups')))
print(' - Cakes       ' + str(totalBrought(allGuests, 'cakes')))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))
print(' - Apple Pies    ' + str(totalBrought(allGuests, 'apple pies')))
```

Inside the `totalBrought()` function, the `for` loop iterates over the key-value pairs in `guests` ❶. Inside the loop, the string of the guest's name is assigned to `k`, and the dictionary of picnic items they're bringing is assigned to `v`. If the item parameter exists as a key in this dictionary, its value (the quantity) is added to `numBrought` ❷. If it does not exist as a key, the `get()` method returns 0 to be added to `numBrought`.

The output of this program looks like this:

```
Number of things being brought:
- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies 1
```

This may seem like such a simple thing to model that you wouldn't need to bother with writing a program to do it. But realize that this same `totalBrought()` function could easily handle a dictionary that contains thousands of guests, each bringing *thousands* of different picnic items. Then having this information in a data structure along with the `totalBrought()` function would save you a lot of time!

You can model things with data structures in whatever way you like, as long as the rest of the code in your program can work with the data model correctly. When you first begin programming, don't worry so much about

the “right” way to model data. As you gain more experience, you may come up with more efficient models, but the important thing is that the data model works for your program’s needs.

Summary

You learned all about dictionaries in this chapter. Lists and dictionaries are values that can contain multiple values, including other lists and dictionaries. Dictionaries are useful because you can map one item (the key) to another (the value), as opposed to lists, which simply contain a series of values in order. Values inside a dictionary are accessed using square brackets just as with lists. Instead of an integer index, dictionaries can have keys of a variety of data types: integers, floats, strings, or tuples. By organizing a program’s values into data structures, you can create representations of real-world objects. You saw an example of this with a tic-tac-toe board.

That just about covers all the basic concepts of Python programming! You’ll continue to learn new concepts throughout the rest of this book, but you now know enough to start writing some useful programs that can automate tasks. You might not think you have enough Python knowledge to do things such as download web pages, update spreadsheets, or send text messages, but that’s where Python modules come in! These modules, written by other programmers, provide functions that make it easy for you to do all these things. So let’s learn how to write real programs to do useful automated tasks.

Practice Questions

1. What does the code for an empty dictionary look like?
2. What does a dictionary value with a key ‘foo’ and a value 42 look like?
3. What is the main difference between a dictionary and a list?
4. What happens if you try to access `spam['foo']` if `spam` is `{'bar': 100}`?
5. If a dictionary is stored in `spam`, what is the difference between the expressions ‘cat’ in `spam` and ‘cat’ in `spam.keys()`?
6. If a dictionary is stored in `spam`, what is the difference between the expressions ‘cat’ in `spam` and ‘cat’ in `spam.values()`?
7. What is a shortcut for the following code?

```
if 'color' not in spam:  
    spam['color'] = 'black'
```

8. What module and function can be used to “pretty print” dictionary values?

Practice Projects

For practice, write programs to do the following tasks.

Fantasy Game Inventory

You are creating a fantasy video game. The data structure to model the player's inventory will be a dictionary where the keys are string values describing the item in the inventory and the value is an integer value detailing how many of that item the player has. For example, the dictionary value `{'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12}` means the player has 1 rope, 6 torches, 42 gold coins, and so on.

Write a function named `displayInventory()` that would take any possible "inventory" and display it like the following:

```
Inventory:  
12 arrow  
42 gold coin  
1 rope  
6 torch  
1 dagger  
Total number of items: 62
```

Hint: You can use a `for` loop to loop through all the keys in a dictionary.

```
# inventory.py  
stuff = {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12}  
  
def displayInventory(inventory):  
    print("Inventory:")  
    item_total = 0  
    for k, v in inventory.items():  
        print(str(v) + ' ' + k)  
        item_total += v  
    print("Total number of items: " + str(item_total))  
  
displayInventory(stuff)
```

List to Dictionary Function for Fantasy Game Inventory

Imagine that a vanquished dragon's loot is represented as a list of strings like this:

```
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
```

Write a function named `addToInventory(inventory, addedItems)`, where the `inventory` parameter is a dictionary representing the player's inventory (like in the previous project) and the `addedItems` parameter is a list like `dragonLoot`.

The `addToInventory()` function should return a dictionary that represents the updated inventory. Note that the `addedItems` list can contain multiples of the same item. Your code could look something like this:

```
def addToInventory(inventory, addedItems):
    # your code goes here

inv = {'gold coin': 42, 'rope': 1}
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
inv = addToInventory(inv, dragonLoot)
displayInventory(inv)
```

The previous program (with your `displayInventory()` function from the previous project) would output the following:

```
Inventory:
45 gold coin
1 rope
1 ruby
1 dagger

Total number of items: 48
```

6

MANIPULATING STRINGS



Text is one of the most common forms of data your programs will handle. You already know how to concatenate two string values together with the `+` operator, but you can do much more than that. You can extract partial strings from string values, add or remove spacing, convert letters to lowercase or uppercase, and check that strings are formatted correctly. You can even write Python code to access the clipboard for copying and pasting text.

In this chapter, you'll learn all this and more. Then you'll work through two different programming projects: a simple password manager and a program to automate the boring chore of formatting pieces of text.

Working with Strings

Let's look at some of the ways Python lets you write, print, and access strings in your code.

String Literals

Typing string values in Python code is fairly straightforward: They begin and end with a single quote. But then how can you use a quote inside a string? Typing `'That is Alice's cat.'` won't work, because Python thinks the string ends after Alice, and the rest (`s cat.'`) is invalid Python code. Fortunately, there are multiple ways to type strings.

Double Quotes

Strings can begin and end with double quotes, just as they do with single quotes. One benefit of using double quotes is that the string can have a single quote character in it. Enter the following into the interactive shell:

```
>>> spam = "That is Alice's cat."
```

Since the string begins with a double quote, Python knows that the single quote is part of the string and not marking the end of the string. However, if you need to use both single quotes and double quotes in the string, you'll need to use escape characters.

Escape Characters

An *escape character* lets you use characters that are otherwise impossible to put into a string. An escape character consists of a backslash (\) followed by the character you want to add to the string. (Despite consisting of two characters, it is commonly referred to as a singular escape character.) For example, the escape character for a single quote is \'. You can use this inside a string that begins and ends with single quotes. To see how escape characters work, enter the following into the interactive shell:

```
>>> spam = 'Say hi to Bob\'s mother.'
```

Python knows that since the single quote in Bob\'s has a backslash, it is not a single quote meant to end the string value. The escape characters \' and \" let you put single quotes and double quotes inside your strings, respectively.

Table 6-1 lists the escape characters you can use.

Table 6-1: Escape Characters

Escape character	Prints as
\'	Single quote
\\"	Double quote
\t	Tab
\n	Newline (line break)
\\\	Backslash

Enter the following into the interactive shell:

```
>>> print("Hello there!\nHow are you?\nI'm doing fine.")  
Hello there!  
How are you?  
I'm doing fine.
```

Raw Strings

You can place an `r` before the beginning quotation mark of a string to make it a raw string. A *raw string* completely ignores all escape characters and prints any backslash that appears in the string. For example, type the following into the interactive shell:

```
>>> print(r'That is Carol\'s cat.')  
That is Carol\''s cat.
```

Because this is a raw string, Python considers the backslash as part of the string and not as the start of an escape character. Raw strings are helpful if you are typing string values that contain many backslashes, such as the strings used for regular expressions described in the next chapter.

Multiline Strings with Triple Quotes

While you can use the `\n` escape character to put a newline into a string, it is often easier to use multiline strings. A multiline string in Python begins and ends with either three single quotes or three double quotes. Any quotes, tabs, or newlines in between the “triple quotes” are considered part of the string. Python’s indentation rules for blocks do not apply to lines inside a multiline string.

Open the file editor and write the following:

```
print('''Dear Alice,  
  
Eve's cat has been arrested for catnapping, cat burglary, and extortion.  
  
Sincerely,  
Bob'''')
```

Save this program as `catnapping.py` and run it. The output will look like this:

```
Dear Alice,  
  
Eve's cat has been arrested for catnapping, cat burglary, and extortion.  
  
Sincerely,  
Bob
```

Notice that the single quote character in Eve's does not need to be escaped. Escaping single and double quotes is optional in raw strings. The following print() call would print identical text but doesn't use a multiline string:

```
print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat  
burglary, and extortion.\n\nSincerely,\nBob')
```

Multiline Comments

While the hash character (#) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines. The following is perfectly valid Python code:

```
"""This is a test Python program.  
Written by Al Sweigart al@inventwithpython.com  
  
This program was designed for Python 3, not Python 2.  
"""  
  
def spam():  
    """This is a multiline comment to help  
    explain what the spam() function does."""  
    print('Hello!')
```

Indexing and Slicing Strings

Strings use indexes and slices the same way lists do. You can think of the string 'Hello world!' as a list and each character in the string as an item with a corresponding index.

'	H	e	l	l	o	w	o	r	l	d	!	'
0	1	2	3	4	5	6	7	8	9	10	11	

The space and exclamation point are included in the character count, so 'Hello world!' is 12 characters long, from H at index 0 to ! at index 11.

Enter the following into the interactive shell:

```
>>> spam = 'Hello world!'  
>>> spam[0]  
'H'  
>>> spam[4]  
'o'  
>>> spam[-1]  
'!'  
>>> spam[0:5]  
'Hello'
```

```
>>> spam[:5]
'Hello'
>>> spam[6:]
'world!'
```

If you specify an index, you'll get the character at that position in the string. If you specify a range from one index to another, the starting index is included and the ending index is not. That's why, if `spam` is `'Hello world!'`, `spam[0:5]` is `'Hello'`. The substring you get from `spam[0:5]` will include everything from `spam[0]` to `spam[4]`, leaving out the space at index 5.

Note that slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable. Try typing the following into the interactive shell:

```
>>> spam = 'Hello world!'
>>> fizz = spam[0:5]
>>> fizz
'Hello'
```

By slicing and storing the resulting substring in another variable, you can have both the whole string and the substring handy for quick, easy access.

The `in` and `not in` Operators with Strings

The `in` and `not in` operators can be used with strings just like with list values. An expression with two strings joined using `in` or `not in` will evaluate to a Boolean `True` or `False`. Enter the following into the interactive shell:

```
>>> 'Hello' in 'Hello World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello World'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

These expressions test whether the first string (the exact string, case sensitive) can be found within the second string.

Useful String Methods

Several string methods analyze strings or create transformed string values. This section describes the methods you'll be using most often.

The `upper()`, `lower()`, `isupper()`, and `islower()` String Methods

The `upper()` and `lower()` string methods return a new string where all the letters in the original string have been converted to uppercase or lowercase, respectively. Nonletter characters in the string remain unchanged. Enter the following into the interactive shell:

```
>>> spam = 'Hello world!'
>>> spam = spam.upper()
>>> spam
'HELLO WORLD!'
>>> spam = spam.lower()
>>> spam
'hello world!'
```

Note that these methods do not change the string itself but return new string values. If you want to change the original string, you have to call `upper()` or `lower()` on the string and then assign the new string to the variable where the original was stored. This is why you must use `spam = spam.upper()` to change the string in `spam` instead of simply `spam.upper()`. (This is just like if a variable `eggs` contains the value `10`. Writing `eggs + 3` does not change the value of `eggs`, but `eggs = eggs + 3` does.)

The `upper()` and `lower()` methods are helpful if you need to make a case-insensitive comparison. The strings '`great`' and '`GREAT`' are not equal to each other. But in the following small program, it does not matter whether the user types `Great`, `GREAT`, or `gREAT`, because the string is first converted to lowercase.

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
    print('I feel great too.')
else:
    print('I hope the rest of your day is good.')
```

When you run this program, the question is displayed, and entering a variation on `great`, such as `GREAT`, will still give the output `I feel great too.` Adding code to your program to handle variations or mistakes in user input, such as inconsistent capitalization, will make your programs easier to use and less likely to fail.

```
How are you?
GREAT
I feel great too.
```

The `isupper()` and `islower()` methods will return a Boolean `True` value if the string has at least one letter and all the letters are uppercase or

lowercase, respectively. Otherwise, the method returns `False`. Enter the following into the interactive shell, and notice what each method call returns:

```
>>> spam = 'Hello world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

Since the `upper()` and `lower()` string methods themselves return strings, you can call string methods on *those* returned string values as well. Expressions that do this will look like a chain of method calls. Enter the following into the interactive shell:

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

The `isX` String Methods

Along with `islower()` and `isupper()`, there are several string methods that have names beginning with the word *is*. These methods return a Boolean value that describes the nature of the string. Here are some common `isX` string methods:

- `isalpha()` returns `True` if the string consists only of letters and is not blank.
- `isalnum()` returns `True` if the string consists only of letters and numbers and is not blank.
- `isdecimal()` returns `True` if the string consists only of numeric characters and is not blank.

- `isspace()` returns `True` if the string consists only of spaces, tabs, and new-lines and is not blank.
- `istitle()` returns `True` if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

Enter the following into the interactive shell:

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> '' .isspace()
True
>>> 'This Is Title Case'.istitle()
True
>>> 'This Is Title Case 123'.istitle()
True
>>> 'This Is not Title Case'.istitle()
False
>>> 'This Is NOT Title Case Either'.istitle()
False
```

The `isX` string methods are helpful when you need to validate user input. For example, the following program repeatedly asks users for their age and a password until they provide valid input. Open a new file editor window and enter this program, saving it as `validateInput.py`:

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')

while True:
    print('Select a new password (letters and numbers only):')
    password = input()
    if password.isalnum():
        break
    print('Passwords can only have letters and numbers.')
```

In the first `while` loop, we ask the user for their age and store their input in `age`. If `age` is a valid (decimal) value, we break out of this first `while` loop and move on to the second, which asks for a password. Otherwise, we inform the user that they need to enter a number and again ask them to

enter their age. In the second while loop, we ask for a password, store the user's input in `password`, and break out of the loop if the input was alphanumeric. If it wasn't, we're not satisfied so we tell the user the password needs to be alphanumeric and again ask them to enter a password.

When run, the program's output looks like this:

```
Enter your age:  
forty two  
Please enter a number for your age.  
Enter your age:  
42  
Select a new password (letters and numbers only):  
secr3t!  
Passwords can only have letters and numbers.  
Select a new password (letters and numbers only):  
secr3t
```

Calling `isdecimal()` and `isalnum()` on variables, we're able to test whether the values stored in those variables are decimal or not, alphanumeric or not. Here, these tests help us reject the input `forty two` and accept `42`, and reject `secr3t!` and accept `secr3t`.

The `startswith()` and `endswith()` String Methods

The `startswith()` and `endswith()` methods return `True` if the string value they are called on begins or ends (respectively) with the string passed to the method; otherwise, they return `False`. Enter the following into the interactive shell:

```
>>> 'Hello world!'.startswith('Hello')  
True  
>>> 'Hello world!'.endswith('world!')  
True  
>>> 'abc123'.startswith('abcdef')  
False  
>>> 'abc123'.endswith('12')  
False  
>>> 'Hello world!'.startswith('Hello world!')  
True  
>>> 'Hello world!'.endswith('Hello world!')  
True
```

These methods are useful alternatives to the `==` equals operator if you need to check only whether the first or last part of the string, rather than the whole thing, is equal to another string.

The `join()` and `split()` String Methods

The `join()` method is useful when you have a list of strings that need to be joined together into a single string value. The `join()` method is called on a

string, gets passed a list of strings, and returns a string. The returned string is the concatenation of each string in the passed-in list. For example, enter the following into the interactive shell:

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

Notice that the string `join()` calls on is inserted between each string of the list argument. For example, when `join(['cats', 'rats', 'bats'])` is called on the `', '` string, the returned string is `'cats, rats, bats'`.

Remember that `join()` is called on a string value and is passed a list value. (It's easy to accidentally call it the other way around.) The `split()` method does the opposite: It's called on a string value and returns a list of strings. Enter the following into the interactive shell:

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

By default, the string `'My name is Simon'` is split wherever whitespace characters such as the space, tab, or newline characters are found. These whitespace characters are not included in the strings in the returned list. You can pass a delimiter string to the `split()` method to specify a different string to split upon. For example, enter the following into the interactive shell:

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

A common use of `split()` is to split a multiline string along the newline characters. Enter the following into the interactive shell:

```
>>> spam = """Dear Alice,
How have you been? I am fine.
There is a container in the fridge
that is labeled "Milk Experiment".

Please do not drink it.
Sincerely,
Bob"""
>>> spam.split('\n')
['Dear Alice,', 'How have you been? I am fine.', 'There is a container in the', 'fridge', 'that is labeled "Milk Experiment".', '', 'Please do not drink it.', 'Sincerely,', 'Bob']
```

Passing `split()` the argument '`\n`' lets us split the multiline string stored in `spam` along the newlines and return a list in which each item corresponds to one line of the string.

Justifying Text with `rjust()`, `ljust()`, and `center()`

The `rjust()` and `ljust()` string methods return a padded version of the string they are called on, with spaces inserted to justify the text. The first argument to both methods is an integer length for the justified string. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(10)
'      Hello'
>>> 'Hello'.rjust(20)
'          Hello'
>>> 'Hello World'.rjust(20)
'          Hello World'
>>> 'Hello'.ljust(10)
'Hello      '
```

`'Hello'.rjust(10)` says that we want to right-justify `'Hello'` in a string of total length 10. `'Hello'` is five characters, so five spaces will be added to its left, giving us a string of 10 characters with `'Hello'` justified right.

An optional second argument to `rjust()` and `ljust()` will specify a fill character other than a space character. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(20, '*')
*****Hello*****
>>> 'Hello'.ljust(20, '-')
Hello-----'
```

The `center()` string method works like `ljust()` and `rjust()` but centers the text rather than justifying it to the left or right. Enter the following into the interactive shell:

```
>>> 'Hello'.center(20)
'      Hello      '
>>> 'Hello'.center(20, '=')
'=====Hello====='
```

These methods are especially useful when you need to print tabular data that has the correct spacing. Open a new file editor window and enter the following code, saving it as `picnicTable.py`:

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))
```

```
picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

In this program, we define a `printPicnic()` method that will take in a dictionary of information and use `center()`, `ljust()`, and `rjust()` to display that information in a neatly aligned table-like format.

The dictionary that we'll pass to `printPicnic()` is `picnicItems`. In `picnicItems`, we have 4 sandwiches, 12 apples, 4 cups, and 8000 cookies. We want to organize this information into two columns, with the name of the item on the left and the quantity on the right.

To do this, we decide how wide we want the left and right columns to be. Along with our dictionary, we'll pass these values to `printPicnic()`.

`printPicnic()` takes in a dictionary, a `leftWidth` for the left column of a table, and a `rightWidth` for the right column. It prints a title, `PICNIC ITEMS`, centered above the table. Then, it loops through the dictionary, printing each key-value pair on a line with the key justified left and padded by periods, and the value justified right and padded by spaces.

After defining `printPicnic()`, we define the dictionary `picnicItems` and call `printPicnic()` twice, passing it different widths for the left and right table columns.

When you run this program, the picnic items are displayed twice. The first time the left column is 12 characters wide, and the right column is 5 characters wide. The second time they are 20 and 6 characters wide, respectively.

```
--PICNIC ITEMS--
sandwiches..      4
apples.....     12
cups.....       4
cookies.....   8000
-----PICNIC ITEMS-----
sandwiches.....      4
apples.....       12
cups.....       4
cookies.....   8000
```

Using `rjust()`, `ljust()`, and `center()` lets you ensure that strings are neatly aligned, even if you aren't sure how many characters long your strings are.

Removing Whitespace with `strip()`, `rstrip()`, and `lstrip()`

Sometimes you may want to strip off whitespace characters (space, tab, and newline) from the left side, right side, or both sides of a string. The `strip()` string method will return a new string without any whitespace

characters at the beginning or end. The `lstrip()` and `rstrip()` methods will remove whitespace characters from the left and right ends, respectively. Enter the following into the interactive shell:

```
>>> spam = 'Hello World'
>>> spam.strip()
'Hello World'
>>> spam.lstrip()
'Hello World '
>>> spam.rstrip()
'Hello World'
```

Optionally, a string argument will specify which characters on the ends should be stripped. Enter the following into the interactive shell:

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

Passing `strip()` the argument 'ampS' will tell it to strip occurrences of a, m, p, and capital S from the ends of the string stored in `spam`. The order of the characters in the string passed to `strip()` does not matter: `strip('amps')` will do the same thing as `strip('mapS')` or `strip('Spam')`.

Copying and Pasting Strings with the pyperclip Module

The `pyperclip` module has `copy()` and `paste()` functions that can send text to and receive text from your computer's clipboard. Sending the output of your program to the clipboard will make it easy to paste it to an email, word processor, or some other software.

`Pyperclip` does not come with Python. To install it, follow the directions for installing third-party modules in Appendix A. After installing the `pyperclip` module, enter the following into the interactive shell:

```
>>> import pyperclip
>>> pyperclip.copy('Hello world!')
>>> pyperclip.paste()
'Hello world!'
```

Of course, if something outside of your program changes the clipboard contents, the `paste()` function will return it. For example, if I copied this sentence to the clipboard and then called `paste()`, it would look like this:

```
>>> pyperclip.paste()
'For example, if I copied this sentence to the clipboard and then called
paste(), it would look like this:'
```

RUNNING PYTHON SCRIPTS OUTSIDE OF IDLE

So far, you've been running your Python scripts using the interactive shell and file editor in IDLE. However, you won't want to go through the inconvenience of opening IDLE and the Python script each time you want to run a script. Fortunately, there are shortcuts you can set up to make running Python scripts easier. The steps are slightly different for Windows, OS X, and Linux, but each is described in Appendix B. Turn to Appendix B to learn how to run your Python scripts conveniently and be able to pass command line arguments to them. (You will not be able to pass command line arguments to your programs using IDLE.)

Project: Password Locker

You probably have accounts on many different websites. It's a bad habit to use the same password for each of them because if any of those sites has a security breach, the hackers will learn the password to all of your other accounts. It's best to use password manager software on your computer that uses one master password to unlock the password manager. Then you can copy any account password to the clipboard and paste it into the website's Password field.

The password manager program you'll create in this example isn't secure, but it offers a basic demonstration of how such programs work.

THE CHAPTER PROJECTS

This is the first "chapter project" of the book. From here on, each chapter will have projects that demonstrate the concepts covered in the chapter. The projects are written in a style that takes you from a blank file editor window to a full, working program. Just like with the interactive shell examples, don't only read the project sections—follow along on your computer!

Step 1: Program Design and Data Structures

You want to be able to run this program with a command line argument that is the account's name—for instance, *email* or *blog*. That account's password will be copied to the clipboard so that the user can paste it into a Password field. This way, the user can have long, complicated passwords without having to memorize them.

Open a new file editor window and save the program as *pw.py*. You need to start the program with a `#! (shebang)` line (see Appendix B) and should also write a comment that briefly describes the program. Since you want to associate each account's name with its password, you can store these as

strings in a dictionary. The dictionary will be the data structure that organizes your account and password data. Make your program look like the following:

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7min1BDDuvMJuxESSKHFhTxFtjVB6',
              'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sd',
              'luggage': '12345'}
```

Step 2: Handle Command Line Arguments

The command line arguments will be stored in the variable `sys.argv`. (See Appendix B for more information on how to use command line arguments in your programs.) The first item in the `sys.argv` list should always be a string containing the program's filename ('`pw.py`'), and the second item should be the first command line argument. For this program, this argument is the name of the account whose password you want. Since the command line argument is mandatory, you display a usage message to the user if they forget to add it (that is, if the `sys.argv` list has fewer than two values in it). Make your program look like the following:

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7min1BDDuvMJuxESSKHFhTxFtjVB6',
              'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sd',
              'luggage': '12345'}

import sys
if len(sys.argv) < 2:
    print('Usage: python pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]      # first command line arg is the account name
```

Step 3: Copy the Right Password

Now that the account name is stored as a string in the variable `account`, you need to see whether it exists in the `PASSWORDS` dictionary as a key. If so, you want to copy the key's value to the clipboard using `pyperclip.copy()`. (Since you're using the `pyperclip` module, you need to import it.) Note that you don't actually *need* the `account` variable; you could just use `sys.argv[1]` everywhere `account` is used in this program. But a variable named `account` is much more readable than something cryptic like `sys.argv[1]`.

Make your program look like the following:

```
#! python3
# pw.py - An insecure password locker program.
```

```
PASSWORDS = {'email': 'F7min1BDDuvMJuxESSKHFhTxFtjVB6',
              'blog': 'VmALvQyKAXiVH5G8v01if1MLZF3sdt',
              'luggage': '12345'}

import sys, pyperclip
if len(sys.argv) < 2:
    print('Usage: py pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]      # first command line arg is the account name

if account in PASSWORDS:
    pyperclip.copy(PASSWORDS[account])
    print('Password for ' + account + ' copied to clipboard.')
else:
    print('There is no account named ' + account)
```

This new code looks in the `PASSWORDS` dictionary for the account name. If the account name is a key in the dictionary, we get the value corresponding to that key, copy it to the clipboard, and print a message saying that we copied the value. Otherwise, we print a message saying there's no account with that name.

That's the complete script. Using the instructions in Appendix B for launching command line programs easily, you now have a fast way to copy your account passwords to the clipboard. You will have to modify the `PASSWORDS` dictionary value in the source whenever you want to update the program with a new password.

Of course, you probably don't want to keep all your passwords in one place where anyone could easily copy them. But you can modify this program and use it to quickly copy regular text to the clipboard. Say you are sending out several emails that have many of the same stock paragraphs in common. You could put each paragraph as a value in the `PASSWORDS` dictionary (you'd probably want to rename the dictionary at this point), and then you would have a way to quickly select and copy one of many standard pieces of text to the clipboard.

On Windows, you can create a batch file to run this program with the WIN-R Run window. (For more about batch files, see Appendix B.) Type the following into the file editor and save the file as `pw.bat` in the `C:\Windows` folder:

```
@py.exe C:\Python34\pw.py %*
@pause
```

With this batch file created, running the password-safe program on Windows is just a matter of pressing WIN-R and typing `pw <account name>`.

Project: Adding Bullets to Wiki Markup

When editing a Wikipedia article, you can create a bulleted list by putting each list item on its own line and placing a star in front. But say you have a really large list that you want to add bullet points to. You could just type those stars at the beginning of each line, one by one. Or you could automate this task with a short Python script.

The *bulletPointAdder.py* script will get the text from the clipboard, add a star and space to the beginning of each line, and then paste this new text to the clipboard. For example, if I copied the following text (for the Wikipedia article “List of Lists of Lists”) to the clipboard:

```
Lists of animals
Lists of aquarium life
Lists of biologists by author abbreviation
Lists of cultivars
```

and then ran the *bulletPointAdder.py* program, the clipboard would then contain the following:

```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

This star-prefixed text is ready to be pasted into a Wikipedia article as a bulleted list.

Step 1: Copy and Paste from the Clipboard

You want the *bulletPointAdder.py* program to do the following:

1. Paste text from the clipboard
2. Do something to it
3. Copy the new text to the clipboard

That second step is a little tricky, but steps 1 and 3 are pretty straightforward: They just involve the `pyperclip.copy()` and `pyperclip.paste()` functions. For now, let’s just write the part of the program that covers steps 1 and 3. Enter the following, saving the program as *bulletPointAdder.py*:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()
```

```
# TODO: Separate lines and add stars.
```

```
pyperclip.copy(text)
```

The `TODO` comment is a reminder that you should complete this part of the program eventually. The next step is to actually implement that piece of the program.

Step 2: Separate the Lines of Text and Add the Star

The call to `pyperclip.paste()` returns all the text on the clipboard as one big string. If we used the “List of Lists of Lists” example, the string stored in `text` would look like this:

```
'Lists of animals\nLists of aquarium life\nLists of biologists by author  
abbreviation\nLists of cultivars'
```

The `\n` newline characters in this string cause it to be displayed with multiple lines when it is printed or pasted from the clipboard. There are many “lines” in this one string value. You want to add a star to the start of each of these lines.

You could write code that searches for each `\n` newline character in the string and then adds the star just after that. But it would be easier to use the `split()` method to return a list of strings, one for each line in the original string, and then add the star to the front of each string in the list.

Make your program look like the following:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):      # loop through all indexes in the "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list

pyperclip.copy(text)
```

We split the text along its newlines to get a list in which each item is one line of the text. We store the list in `lines` and then loop through the items in `lines`. For each line, we add a star and a space to the start of the line. Now each string in `lines` begins with a star.

Step 3: Join the Modified Lines

The `lines` list now contains modified lines that start with stars. But `pyperclip.copy()` is expecting a single string value, not a list of string values. To make this single string value, pass `lines` into the `join()` method to get a single string joined from the list's strings. Make your program look like the following:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):
    # loop through all indexes for "lines" list
    lines[i] = '*' + lines[i] # add star to each string in "lines" list
text = '\n'.join(lines)
pyperclip.copy(text)
```

When this program is run, it replaces the text on the clipboard with text that has stars at the start of each line. Now the program is complete, and you can try running it with text copied to the clipboard.

Even if you don't need to automate this specific task, you might want to automate some other kind of text manipulation, such as removing trailing spaces from the end of lines or converting text to uppercase or lowercase. Whatever your needs, you can use the clipboard for input and output.

Summary

Text is a common form of data, and Python comes with many helpful string methods to process the text stored in string values. You will make use of indexing, slicing, and string methods in almost every Python program you write.

The programs you are writing now don't seem too sophisticated—they don't have graphical user interfaces with images and colorful text. So far, you're displaying text with `print()` and letting the user enter text with `input()`. However, the user can quickly enter large amounts of text through the clipboard. This ability provides a useful avenue for writing programs that manipulate massive amounts of text. These text-based programs might not have flashy windows or graphics, but they can get a lot of useful work done quickly.

Another way to manipulate large amounts of text is reading and writing files directly off the hard drive. You'll learn how to do this with Python in the next chapter.

Practice Questions

1. What are escape characters?
2. What do the \n and \t escape characters represent?
3. How can you put a \ backslash character in a string?
4. The string value "Howl's Moving Castle" is a valid string. Why isn't it a problem that the single quote character in the word Howl's isn't escaped?
5. If you don't want to put \n in your string, how can you write a string with newlines in it?
6. What do the following expressions evaluate to?
 - 'Hello world!'[1]
 - 'Hello world!'[0:5]
 - 'Hello world!'[:5]
 - 'Hello world!'[3:]
7. What do the following expressions evaluate to?
 - 'Hello'.upper()
 - 'Hello'.upper().isupper()
 - 'Hello'.upper().lower()
8. What do the following expressions evaluate to?
 - 'Remember, remember, the fifth of November.'.split()
 - '-'.join('There can be only one.'.split())
9. What string methods can you use to right-justify, left-justify, and center a string?
10. How can you trim whitespace characters from the beginning or end of a string?

Practice Project

For practice, write a program that does the following.

Table Printer

Write a function named `printTable()` that takes a list of lists of strings and displays it in a well-organized table with each column right-justified. Assume that all the inner lists will contain the same number of strings. For example, the value could look like this:

```
tableData = [['apples', 'oranges', 'cherries', 'banana'],
             ['Alice', 'Bob', 'Carol', 'David'],
             ['dogs', 'cats', 'moose', 'goose']]
```

Your `printTable()` function would print the following:

```
apples Alice dogs
oranges Bob cats
cherries Carol moose
banana David goose
```

Hint: Your code will first have to find the longest string in each of the inner lists so that the whole column can be wide enough to fit all the strings. You can store the maximum width of each column as a list of integers. The `printTable()` function can begin with `colWidths = [0] * len(tableData)`, which will create a list containing the same number of 0 values as the number of inner lists in `tableData`. That way, `colWidths[0]` can store the width of the longest string in `tableData[0]`, `colWidths[1]` can store the width of the longest string in `tableData[1]`, and so on. You can then find the largest value in the `colWidths` list to find out what integer width to pass to the `rjust()` string method.

PART II

AUTOMATING TASKS

7

PATTERN MATCHING WITH REGULAR EXPRESSIONS



You may be familiar with searching for text by pressing CTRL-F and typing in the words you're looking for. *Regular expressions* go one step further: They allow you to specify a *pattern* of text to search for. You may not know a business's exact phone number, but if you live in the United States or Canada, you know it will be three digits, followed by a hyphen, and then four more digits (and optionally, a three-digit area code at the start). This is how you, as a human, know a phone number when you see it: 415-555-1234 is a phone number, but 4,155,551,234 is not.

Regular expressions are helpful, but not many non-programmers know about them even though most modern text editors and word processors, such as Microsoft Word or OpenOffice, have find and find-and-replace features that can search based on regular expressions. Regular expressions are huge time-savers, not just for software users but also for

programmers. In fact, tech writer Cory Doctorow argues that even before teaching programming, we should be teaching regular expressions:

“Knowing [regular expressions] can mean the difference between solving a problem in 3 steps and solving it in 3,000 steps. When you’re a nerd, you forget that the problems you solve with a couple keystrokes can take other people days of tedious, error-prone work to slog through.”¹

In this chapter, you’ll start by writing a program to find text patterns *without* using regular expressions and then see how to use regular expressions to make the code much less bloated. I’ll show you basic matching with regular expressions and then move on to some more powerful features, such as string substitution and creating your own character classes. Finally, at the end of the chapter, you’ll write a program that can automatically extract phone numbers and email addresses from a block of text.

Finding Patterns of Text Without Regular Expressions

Say you want to find a phone number in a string. You know the pattern: three numbers, a hyphen, three numbers, a hyphen, and four numbers. Here’s an example: 415-555-4242.

Let’s use a function named `isPhoneNumber()` to check whether a string matches this pattern, returning either `True` or `False`. Open a new file editor window and enter the following code; then save the file as `isPhoneNumber.py`:

```
def isPhoneNumber(text):
    ❶    if len(text) != 12:
        return False
    for i in range(0, 3):
        ❷        if not text[i].isdecimal():
            return False
    ❸    if text[3] != '-':
        return False
    for i in range(4, 7):
        ❹        if not text[i].isdecimal():
            return False
    ❺    if text[7] != '-':
        return False
    for i in range(8, 12):
        ❻        if not text[i].isdecimal():
            return False
    ❼    return True

print('415-555-4242 is a phone number:')
print(isPhoneNumber('415-555-4242'))
print('Moshi moshi is a phone number:')
print(isPhoneNumber('Moshi moshi'))
```

1. Cory Doctorow, “Here’s what ICT should really teach kids: how to do regular expressions,” *Guardian*, December 4, 2012, <http://www.theguardian.com/technology/2012/dec/04/ict-teach-kids-regular-expressions/>.

When this program is run, the output looks like this:

```
415-555-4242 is a phone number:  
True  
Moshi moshi is a phone number:  
False
```

The `isPhoneNumber()` function has code that does several checks to see whether the string in `text` is a valid phone number. If any of these checks fail, the function returns `False`. First the code checks that the string is exactly 12 characters ❶. Then it checks that the area code (that is, the first three characters in `text`) consists of only numeric characters ❷. The rest of the function checks that the string follows the pattern of a phone number: The number must have the first hyphen after the area code ❸, three more numeric characters ❹, then another hyphen ❺, and finally four more numbers ❻. If the program execution manages to get past all the checks, it returns `True` ❼.

Calling `isPhoneNumber()` with the argument '415-555-4242' will return `True`. Calling `isPhoneNumber()` with 'Moshi moshi' will return `False`; the first test fails because 'Moshi moshi' is not 12 characters long.

You would have to add even more code to find this pattern of text in a larger string. Replace the last four `print()` function calls in `isPhoneNumber.py` with the following:

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'  
for i in range(len(message)):  
❶    chunk = message[i:i+12]  
❷    if isPhoneNumber(chunk):  
        print('Phone number found: ' + chunk)  
print('Done')
```

When this program is run, the output will look like this:

```
Phone number found: 415-555-1011  
Phone number found: 415-555-9999  
Done
```

On each iteration of the `for` loop, a new chunk of 12 characters from `message` is assigned to the variable `chunk` ❶. For example, on the first iteration, `i` is 0, and `chunk` is assigned `message[0:12]` (that is, the string 'Call me at 4'). On the next iteration, `i` is 1, and `chunk` is assigned `message[1:13]` (the string 'all me at 41').

You pass `chunk` to `isPhoneNumber()` to see whether it matches the phone number pattern ❷, and if so, you print the chunk.

Continue to loop through `message`, and eventually the 12 characters in `chunk` will be a phone number. The loop goes through the entire string, testing each 12-character piece and printing any chunk it finds that satisfies `isPhoneNumber()`. Once we're done going through `message`, we print `Done`.

While the string in `message` is short in this example, it could be millions of characters long and the program would still run in less than a second. A similar program that finds phone numbers using regular expressions would also run in less than a second, but regular expressions make it quicker to write these programs.

Finding Patterns of Text with Regular Expressions

The previous phone number–finding program works, but it uses a lot of code to do something limited: The `isPhoneNumber()` function is 17 lines but can find only one pattern of phone numbers. What about a phone number formatted like 415.555.4242 or (415) 555-4242? What if the phone number had an extension, like 415-555-4242 x99? The `isPhoneNumber()` function would fail to validate them. You could add yet more code for these additional patterns, but there is an easier way.

Regular expressions, called *regexes* for short, are descriptions for a pattern of text. For example, a `\d` in a regex stands for a digit character—that is, any single numeral 0 to 9. The regex `\d\d\d-\d\d\d-\d\d\d` is used by Python to match the same text the previous `isPhoneNumber()` function did: a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers. Any other string would not match the `\d\d\d-\d\d\d-\d\d\d` regex.

But regular expressions can be much more sophisticated. For example, adding a 3 in curly brackets (`{3}`) after a pattern is like saying, “Match this pattern three times.” So the slightly shorter regex `\d{3}-\d{3}-\d{4}` also matches the correct phone number format.

Creating Regex Objects

All the regex functions in Python are in the `re` module. Enter the following into the interactive shell to import this module:

```
>>> import re
```

NOTE

Most of the examples that follow in this chapter will require the `re` module, so remember to import it at the beginning of any script you write or any time you restart IDLE. Otherwise, you'll get a `NameError: name 're' is not defined` error message.

Passing a string value representing your regular expression to `re.compile()` returns a Regex pattern object (or simply, a `Regex` object).

To create a `Regex` object that matches the phone number pattern, enter the following into the interactive shell. (Remember that `\d` means “a digit character” and `\d\d\d-\d\d\d-\d\d\d` is the regular expression for the correct phone number pattern.)

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

Now the phoneNumRegex variable contains a Regex object.

PASSING RAW STRINGS TO RE.COMPILE()

Remember that escape characters in Python use the backslash (\). The string value '\n' represents a single newline character, not a backslash followed by a lowercase *n*. You need to enter the escape character \\ to print a single backslash. So '\\n' is the string that represents a backslash followed by a lowercase *n*. However, by putting an *r* before the first quote of the string value, you can mark the string as a *raw string*, which does not escape characters.

Since regular expressions frequently use backslashes in them, it is convenient to pass raw strings to the `re.compile()` function instead of typing extra backslashes. Typing `r'\d\d\d-\d\d\d-\d\d\d\d'` is much easier than typing `'\\d\\d\\d-\\d\\d\\d-\\d\\d\\d\\d'`.

Matching Regex Objects

A Regex object's `search()` method searches the string it is passed for any matches to the regex. The `search()` method will return `None` if the regex pattern is not found in the string. If the pattern *is* found, the `search()` method returns a `Match` object. `Match` objects have a `group()` method that will return the actual matched text from the searched string. (I'll explain groups shortly.) For example, enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found: ' + mo.group())
Phone number found: 415-555-4242
```

The `mo` variable name is just a generic name to use for `Match` objects. This example might seem complicated at first, but it is much shorter than the earlier `isPhoneNumber.py` program and does the same thing.

Here, we pass our desired pattern to `re.compile()` and store the resulting Regex object in `phoneNumRegex`. Then we call `search()` on `phoneNumRegex` and pass `search()` the string we want to search for a match. The result of the search gets stored in the variable `mo`. In this example, we know that our pattern will be found in the string, so we know that a `Match` object will be returned. Knowing that `mo` contains a `Match` object and not the null value `None`, we can call `group()` on `mo` to return the match. Writing `mo.group()` inside our `print` statement displays the whole match, 415-555-4242.

Review of Regular Expression Matching

While there are several steps to using regular expressions in Python, each step is fairly simple.

1. Import the regex module with `import re`.
2. Create a Regex object with the `re.compile()` function. (Remember to use a raw string.)
3. Pass the string you want to search into the Regex object's `search()` method. This returns a Match object.
4. Call the Match object's `group()` method to return a string of the actual matched text.

NOTE

While I encourage you to enter the example code into the interactive shell, you should also make use of web-based regular expression testers, which can show you exactly how a regex matches a piece of text that you enter. I recommend the tester at <http://regexpal.com/>.

More Pattern Matching with Regular Expressions

Now that you know the basic steps for creating and finding regular expression objects with Python, you're ready to try some of their more powerful pattern-matching capabilities.

Grouping with Parentheses

Say you want to separate the area code from the rest of the phone number. Adding parentheses will create *groups* in the regex: `(\d\d\d)-(\d\d\d-\d\d\d\d)`. Then you can use the `group()` match object method to grab the matching text from just one group.

The first set of parentheses in a regex string will be group 1. The second set will be group 2. By passing the integer 1 or 2 to the `group()` match object method, you can grab different parts of the matched text. Passing 0 or nothing to the `group()` method will return the entire matched text. Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

If you would like to retrieve all the groups at once, use the `groups()` method—note the plural form for the name.

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

Since `mo.groups()` returns a tuple of multiple values, you can use the multiple-assignment trick to assign each value to a separate variable, as in the previous `areaCode, mainNumber = mo.groups()` line.

Parentheses have a special meaning in regular expressions, but what do you do if you need to match a parenthesis in your text? For instance, maybe the phone numbers you are trying to match have the area code set in parentheses. In this case, you need to escape the (and) characters with a back-slash. Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d\d)')
>>> mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

The \(\ and \) escape characters in the raw string passed to `re.compile()` will match actual parenthesis characters.

Matching Multiple Groups with the Pipe

The | character is called a *pipe*. You can use it anywhere you want to match one of many expressions. For example, the regular expression `r'Batman|Tina Fey'` will match either 'Batman' or 'Tina Fey'.

When *both* Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the Match object. Enter the following into the interactive shell:

```
>>> heroRegex = re.compile (r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey.')
>>> mo1.group()
'Batman'

>>> mo2 = heroRegex.search('Tina Fey and Batman.')
>>> mo2.group()
'Tina Fey'
```

NOTE

You can find all matching occurrences with the `findall()` method that's discussed in "The `findall()` Method" on page 157.

You can also use the pipe to match one of several patterns as part of your regex. For example, say you wanted to match any of the strings 'Batman', 'Batmobile', 'Batcopter', and 'Batbat'. Since all these strings start with Bat, it would be nice if you could specify that prefix only once. This can be done with parentheses. Enter the following into the interactive shell:

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1)
'mobile'
```

The method call `mo.group()` returns the full matched text 'Batmobile', while `mo.group(1)` returns just the part of the matched text inside the first parentheses group, 'mobile'. By using the pipe character and grouping parentheses, you can specify several alternative patterns you would like your regex to match.

If you need to match an actual pipe character, escape it with a backslash, like `\|`.

Optional Matching with the Question Mark

Sometimes there is a pattern that you want to match only optionally. That is, the regex should find a match whether or not that bit of text is there. The `?` character flags the group that precedes it as an optional part of the pattern. For example, enter the following into the interactive shell:

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

The `(wo)?` part of the regular expression means that the pattern `wo` is an optional group. The regex will match text that has zero instances or one instance of `wo` in it. This is why the regex matches both 'Batwoman' and 'Batman'.

Using the earlier phone number example, you can make the regex look for phone numbers that do or do not have an area code. Enter the following into the interactive shell:

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>> mo1 = phoneRegex.search('My number is 415-555-4242')
>>> mo1.group()
'415-555-4242'
```

```
>>> mo2 = phoneRegex.search('My number is 555-4242')
>>> mo2.group()
'555-4242'
```

You can think of the ? as saying, “Match zero or one of the group preceding this question mark.”

If you need to match an actual question mark character, escape it with \?.

Matching Zero or More with the Star

The * (called the *star* or *asterisk*) means “match zero or more”—the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again. Let’s look at the Batman example again.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'

>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

For ‘Batman’, the (wo)* part of the regex matches zero instances of wo in the string; for ‘Batwoman’, the (wo)* matches one instance of wo; and for ‘Batwowowowoman’, (wo)* matches four instances of wo.

If you need to match an actual star character, prefix the star in the regular expression with a backslash, *.

Matching One or More with the Plus

While * means “match zero or more,” the + (or *plus*) means “match one or more.” Unlike the star, which does not require its group to appear in the matched string, the group preceding a plus must appear *at least once*. It is not optional. Enter the following into the interactive shell, and compare it with the star regexes in the previous section:

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'

>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'
```

```
>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

The regex Bat(wo)+man will not match the string 'The Adventures of Batman' because at least one wo is required by the plus sign.

If you need to match an actual plus sign character, prefix the plus sign with a backslash to escape it: \+.

Matching Specific Repetitions with Curly Brackets

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets. For example, the regex (Ha){3} will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets. For example, the regex (Ha){3,5} will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'.

You can also leave out the first or second number in the curly brackets to leave the minimum or maximum unbounded. For example, (Ha){3,} will match three or more instances of the (Ha) group, while (Ha){,5} will match zero to five instances. Curly brackets can help make your regular expressions shorter. These two regular expressions match identical patterns:

```
(Ha){3}
(Ha)(Ha)(Ha)
```

And these two regular expressions also match identical patterns:

```
(Ha){3,5}
((Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha)(Ha))
```

Enter the following into the interactive shell:

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'

>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

Here, (Ha){3} matches 'HaHaHa' but not 'Ha'. Since it doesn't match 'Ha', search() returns None.

Greedy and Nongreedy Matching

Since (Ha){3,5} can match three, four, or five instances of Ha in the string 'HaHaHaHaHa', you may wonder why the Match object's call to group() in the

previous curly bracket example returns 'HaHaHaHaHa' instead of the shorter possibilities. After all, 'HaHaHa' and 'HaHaHaHa' are also valid matches of the regular expression (Ha){3,5}.

Python's regular expressions are *greedy* by default, which means that in ambiguous situations they will match the longest string possible. The *nongreedy* version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark.

Enter the following into the interactive shell, and notice the difference between the greedy and nongreedy forms of the curly brackets searching the same string:

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'

>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

Note that the question mark can have two meanings in regular expressions: declaring a nongreedy match or flagging an optional group. These meanings are entirely unrelated.

The `.findall()` Method

In addition to the `search()` method, Regex objects also have a `.findall()` method. While `search()` will return a `Match` object of the *first* matched text in the searched string, the `.findall()` method will return the strings of *every* match in the searched string. To see how `search()` returns a `Match` object only on the first instance of matching text, enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>> mo.group()
'415-555-9999'
```

On the other hand, `.findall()` will not return a `Match` object but a list of strings—as long as there are no groups in the regular expression. Each string in the list is a piece of the searched text that matched the regular expression. Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

If there *are* groups in the regular expression, then `.findall()` will return a list of tuples. Each tuple represents a found match, and its items are the

matched strings for each group in the regex. To see `.findall()` in action, enter the following into the interactive shell (notice that the regular expression being compiled now has groups in parentheses):

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[('415', '555', '1122'), ('212', '555', '0000')]
```

To summarize what the `.findall()` method returns, remember the following:

1. When called on a regex with no groups, such as `\d\d\d-\d\d\d-\d\d\d\d`, the method `.findall()` returns a list of string matches, such as `['415-555-9999', '212-555-0000']`.
2. When called on a regex that has groups, such as `(\d\d\d)-(\d\d\d)-(\d\d\d\d)`, the method `.findall()` returns a list of tuples of strings (one string for each group), such as `[('415', '555', '1122'), ('212', '555', '0000')]`.

Character Classes

In the earlier phone number regex example, you learned that `\d` could stand for any numeric digit. That is, `\d` is shorthand for the regular expression `(0|1|2|3|4|5|6|7|8|9)`. There are many such *shorthand character classes*, as shown in Table 7-1.

Table 7-1: Shorthand Codes for Common Character Classes

Shorthand character class	Represents
<code>\d</code>	Any numeric digit from 0 to 9.
<code>\D</code>	Any character that is <i>not</i> a numeric digit from 0 to 9.
<code>\w</code>	Any letter, numeric digit, or the underscore character. (Think of this as matching “word” characters.)
<code>\W</code>	Any character that is <i>not</i> a letter, numeric digit, or the underscore character.
<code>\s</code>	Any space, tab, or newline character. (Think of this as matching “space” characters.)
<code>\S</code>	Any character that is <i>not</i> a space, tab, or newline.

Character classes are nice for shortening regular expressions. The character class `[0-5]` will match only the numbers 0 to 5; this is much shorter than typing `(0|1|2|3|4|5)`.

For example, enter the following into the interactive shell:

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7
swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6
geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

The regular expression `\d+\s\w+` will match text that has one or more numeric digits (`\d+`), followed by a whitespace character (`\s`), followed by one or more letter/digit/underscore characters (`\w+`). The `.findall()` method returns all matching strings of the regex pattern in a list.

Making Your Own Character Classes

There are times when you want to match a set of characters but the short-hand character classes (`\d`, `\w`, `\s`, and so on) are too broad. You can define your own character class using square brackets. For example, the character class `[aeiouAEIOU]` will match any vowel, both lowercase and uppercase. Enter the following into the interactive shell:

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class `[a-zA-Z0-9]` will match all lowercase letters, uppercase letters, and numbers.

Note that inside the square brackets, the normal regular expression symbols are not interpreted as such. This means you do not need to escape the `.`, `*`, `?`, or `()` characters with a preceding backslash. For example, the character class `[0-5.]` will match digits 0 to 5 and a period. You do not need to write it as `[0-5\\.]`.

By placing a caret character (^) just after the character class's opening bracket, you can make a *negative character class*. A negative character class will match all the characters that are *not* in the character class. For example, enter the following into the interactive shell:

```
>>> consonantRegex = re.compile(r'[^aeiouAEIOU]')
>>> consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')
['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.', ' ', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

Now, instead of matching every vowel, we're matching every character that isn't a vowel.

The Caret and Dollar Sign Characters

You can also use the caret symbol (^) at the start of a regex to indicate that a match must occur at the *beginning* of the searched text. Likewise, you can put a dollar sign (\$) at the end of the regex to indicate the string must *end* with this regex pattern. And you can use the ^ and \$ together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.

For example, the `r'^Hello'` regular expression string matches strings that begin with '`Hello`'. Enter the following into the interactive shell:

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')
<sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

The `r'\d$'` regular expression string matches strings that end with a numeric character from 0 to 9. Enter the following into the interactive shell:

```
>>> endsWithNumber = re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')
<sre.SRE_Match object; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None
True
```

The `r'^\d+$'` regular expression string matches strings that both begin and end with one or more numeric characters. Enter the following into the interactive shell:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')
>>> wholeStringIsNum.search('1234567890')
<sre.SRE_Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12 34567890') == None
True
```

The last two `search()` calls in the previous interactive shell example demonstrate how the entire string must match the regex if `^` and `$` are used.

I always confuse the meanings of these two symbols, so I use the mnemonic “Carrots cost dollars” to remind myself that the caret comes first and the dollar sign comes last.

The Wildcard Character

The `.` (or *dot*) character in a regular expression is called a *wildcard* and will match any character except for a newline. For example, enter the following into the interactive shell:

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

Remember that the dot character will match just one character, which is why the match for the text `flat` in the previous example matched only `lat`. To match an actual dot, escape the dot with a backslash: \..

Matching Everything with Dot-Star

Sometimes you will want to match everything and anything. For example, say you want to match the string 'First Name:', followed by any and all text, followed by 'Last Name:', and then followed by anything again. You can use the dot-star `(.*)` to stand in for that "anything." Remember that the dot character means "any single character except the newline," and the star character means "zero or more of the preceding character."

Enter the following into the interactive shell:

```
>>> nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')
>>> mo.group(1)
'Al'
>>> mo.group(2)
'Sweigart'
```

The dot-star uses *greedy* mode: It will always try to match as much text as possible. To match any and all text in a *nongreedy* fashion, use the dot, star, and question mark `(.*?)`. Like with curly brackets, the question mark tells Python to match in a nongreedy way.

Enter the following into the interactive shell to see the difference between the greedy and nongreedy versions:

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>

>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man> for dinner.>'
```

Both regexes roughly translate to "Match an opening angle bracket, followed by anything, followed by a closing angle bracket." But the string '`<To serve man> for dinner.>`' has two possible matches for the closing angle bracket. In the nongreedy version of the regex, Python matches the shortest possible string: '`<To serve man>`'. In the greedy version, Python matches the longest possible string: '`<To serve man> for dinner.>`'.

Matching Newlines with the Dot Character

The dot-star will match everything except a newline. By passing `re.DOTALL` as the second argument to `re.compile()`, you can make the dot character match *all* characters, including the newline character.

Enter the following into the interactive shell:

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'

>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

The regex `noNewlineRegex`, which did not have `re.DOTALL` passed to the `re.compile()` call that created it, will match everything only up to the first newline character, whereas `newlineRegex`, which *did* have `re.DOTALL` passed to `re.compile()`, matches everything. This is why the `newlineRegex.search()` call matches the full string, including its newline characters.

Review of Regex Symbols

This chapter covered a lot of notation, so here's a quick review of what you learned:

- The `?` matches zero or one of the preceding group.
- The `*` matches zero or more of the preceding group.
- The `+` matches one or more of the preceding group.
- The `{n}` matches exactly *n* of the preceding group.
- The `{n,}` matches *n* or more of the preceding group.
- The `{,m}` matches 0 to *m* of the preceding group.
- The `{n,m}` matches at least *n* and at most *m* of the preceding group.
- `{n,m}?` or `*?` or `+`? performs a nongreedy match of the preceding group.
- `^spam` means the string must begin with *spam*.
- `spam$` means the string must end with *spam*.
- The `.` matches any character, except newline characters.
- `\d`, `\w`, and `\s` match a digit, word, or space character, respectively.
- `\D`, `\W`, and `\S` match anything except a digit, word, or space character, respectively.
- `[abc]` matches any character between the brackets (such as *a*, *b*, or *c*).
- `[^abc]` matches any character that isn't between the brackets.

Case-Insensitive Matching

Normally, regular expressions match text with the exact casing you specify. For example, the following regexes match completely different strings:

```
>>> regex1 = re.compile('RoboCop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('rob0cop')
>>> regex4 = re.compile('RobocOp')
```

But sometimes you care only about matching the letters without worrying whether they're uppercase or lowercase. To make your regex case-insensitive, you can pass `re.IGNORECASE` or `re.I` as a second argument to `re.compile()`. Enter the following into the interactive shell:

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('RoboCop is part man, part machine, all cop.').group()
'RoboCop'

>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'

>>> robocop.search('Al, why does your programming book talk about robocop so much?').group()
'robocop'
```

Substituting Strings with the `sub()` Method

Regular expressions can not only find text patterns but can also substitute new text in place of those patterns. The `sub()` method for Regex objects is passed two arguments. The first argument is a string to replace any matches. The second is the string for the regular expression. The `sub()` method returns a string with the substitutions applied.

For example, enter the following into the interactive shell:

```
>>> namesRegex = re.compile(r'Agent \w+')
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
'CENSORED gave the secret documents to CENSORED.'
```

Sometimes you may need to use the matched text itself as part of the substitution. In the first argument to `sub()`, you can type `\1`, `\2`, `\3`, and so on, to mean “Enter the text of group 1, 2, 3, and so on, in the substitution.”

For example, say you want to censor the names of the secret agents by showing just the first letters of their names. To do this, you could use the regex `Agent (\w)\w*` and pass `r'\1****'` as the first argument to `sub()`. The `\1` in that string will be replaced by whatever text was matched by group 1—that is, the `(\w)` group of the regular expression.

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')
>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent
Eve knew Agent Bob was a double agent.')
Agent **** told Agent **** that Eve **** knew Agent **** was a double agent.'
```

Managing Complex Regexes

Regular expressions are fine if the text pattern you need to match is simple. But matching complicated text patterns might require long, convoluted regular expressions. You can mitigate this by telling the `re.compile()` function to ignore whitespace and comments inside the regular expression string. This “verbose mode” can be enabled by passing the variable `re.VERBOSE` as the second argument to `re.compile()`.

Now instead of a hard-to-read regular expression like this:

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?\s|-|\.)?\d{3}(\s|-|\.)\d{4}
(\s*(ext|x|ext.)\s*\d{2,5})?)')
```

you can spread the regular expression over multiple lines with comments like this:

```
phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?          # area code
    (\s|-|\.)?                # separator
    \d{3}                      # first 3 digits
    (\s|-|\.)?                # separator
    \d{4}                      # last 4 digits
    (\s*(ext|x|ext.)\s*\d{2,5})? # extension
)''', re.VERBOSE)
```

Note how the previous example uses the triple-quote syntax ('') to create a multiline string so that you can spread the regular expression definition over many lines, making it much more legible.

The comment rules inside the regular expression string are the same as regular Python code: The `#` symbol and everything after it to the end of the line are ignored. Also, the extra spaces inside the multiline string for the regular expression are not considered part of the text pattern to be matched. This lets you organize the regular expression so it’s easier to read.

Combining `re.IGNORECASE`, `re.DOTALL`, and `re.VERBOSE`

What if you want to use `re.VERBOSE` to write comments in your regular expression but also want to use `re.IGNORECASE` to ignore capitalization? Unfortunately, the `re.compile()` function takes only a single value as its second argument. You can get around this limitation by combining the `re.IGNORECASE`, `re.DOTALL`, and `re.VERBOSE` variables using the pipe character (`|`), which in this context is known as the *bitwise or* operator.

So if you want a regular expression that's case-insensitive *and* includes newlines to match the dot character, you would form your `re.compile()` call like this:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

All three options for the second argument will look like this:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)
```

This syntax is a little old-fashioned and originates from early versions of Python. The details of the bitwise operators are beyond the scope of this book, but check out the resources at <http://nostarch.com/automatestuff/> for more information. You can also pass other options for the second argument; they're uncommon, but you can read more about them in the resources, too.

Project: Phone Number and Email Address Extractor

Say you have the boring task of finding every phone number and email address in a long web page or document. If you manually scroll through the page, you might end up searching for a long time. But if you had a program that could search the text in your clipboard for phone numbers and email addresses, you could simply press `CTRL-A` to select all the text, press `CTRL-C` to copy it to the clipboard, and then run your program. It could replace the text on the clipboard with just the phone numbers and email addresses it finds.

Whenever you're tackling a new project, it can be tempting to dive right into writing code. But more often than not, it's best to take a step back and consider the bigger picture. I recommend first drawing up a high-level plan for what your program needs to do. Don't think about the actual code yet—you can worry about that later. Right now, stick to broad strokes.

For example, your phone and email address extractor will need to do the following:

- Get the text off the clipboard.
- Find all phone numbers and email addresses in the text.
- Paste them onto the clipboard.

Now you can start thinking about how this might work in code. The code will need to do the following:

- Use the `pyperclip` module to copy and paste strings.
- Create two regexes, one for matching phone numbers and the other for matching email addresses.
- Find all matches, not just the first match, of both regexes.
- Neatly format the matched strings into a single string to paste.
- Display some kind of message if no matches were found in the text.

This list is like a road map for the project. As you write the code, you can focus on each of these steps separately. Each step is fairly manageable and expressed in terms of things you already know how to do in Python.

Step 1: Create a Regex for Phone Numbers

First, you have to create a regular expression to search for phone numbers. Create a new file, enter the following, and save it as *phoneAndEmail.py*:

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?
    (\s|-|\.)?
    (\d{3})
    (\s|-|\.)
    (\d{4})
    (\s*(ext|x|ext.)\s*(\d{2,5}))?
)''', re.VERBOSE)

# TODO: Create email regex.

# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.
```

The `TODO` comments are just a skeleton for the program. They'll be replaced as you write the actual code.

The phone number begins with an *optional* area code, so the area code group is followed with a question mark. Since the area code can be just three digits (that is, `\d{3}`) or three digits within parentheses (that is, `\(\d{3}\)`), you should have a pipe joining those parts. You can add the regex comment `# Area code` to this part of the multiline string to help you remember what `(\d{3}|\(\d{3}\))?` is supposed to match.

The phone number separator character can be a space (`\s`), hyphen (-), or period (.), so these parts should also be joined by pipes. The next few parts of the regular expression are straightforward: three digits, followed by another separator, followed by four digits. The last part is an optional extension made up of any number of spaces followed by ext, x, or ext., followed by two to five digits.

Step 2: Create a Regex for Email Addresses

You will also need a regular expression that can match email addresses. Make your program look like the following:

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.
```

```

import pyperclip, re

phoneRegex = re.compile(r'''(
--snip--

# Create email regex.
emailRegex = re.compile(r'''(
❶    [a-zA-Z0-9._%+-]+      # username
❷    @                      # @ symbol
❸    [a-zA-Z0-9.-]+         # domain name
        (\.[a-zA-Z]{2,4})    # dot-something
)''', re.VERBOSE)

# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.

```

The username part of the email address ❶ is one or more characters that can be any of the following: lowercase and uppercase letters, numbers, a dot, an underscore, a percent sign, a plus sign, or a hyphen. You can put all of these into a character class: [a-zA-Z0-9._%+-].

The domain and username are separated by an @ symbol ❷. The domain name ❸ has a slightly less permissive character class with only letters, numbers, periods, and hyphens: [a-zA-Z0-9.-]. And last will be the “dot-com” part (technically known as the *top-level domain*), which can really be dot-anything. This is between two and four characters.

The format for email addresses has a lot of weird rules. This regular expression won’t match every possible valid email address, but it’ll match almost any typical email address you’ll encounter.

Step 3: Find All Matches in the Clipboard Text

Now that you have specified the regular expressions for phone numbers and email addresses, you can let Python’s `re` module do the hard work of finding all the matches on the clipboard. The `pyperclip.paste()` function will get a string value of the text on the clipboard, and the `.findall()` regex method will return a list of tuples.

Make your program look like the following:

```

#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''(
--snip--

# Find matches in clipboard text.
text = str(pyperclip.paste())

```

```
❶ matches = []
❷ for groups in phoneRegex.findall(text):
    phoneNum = '-'.join([groups[1], groups[3], groups[5]])
    if groups[8] != '':
        phoneNum += ' x' + groups[8]
    matches.append(phoneNum)
❸ for groups in emailRegex.findall(text):
    matches.append(groups[0])

# TODO: Copy results to the clipboard.
```

There is one tuple for each match, and each tuple contains strings for each group in the regular expression. Remember that group 0 matches the entire regular expression, so the group at index 0 of the tuple is the one you are interested in.

As you can see at ❶, you'll store the matches in a list variable named `matches`. It starts off as an empty list, and a couple for loops. For the email addresses, you append group 0 of each match ❸. For the matched phone numbers, you don't want to just append group 0. While the program *detects* phone numbers in several formats, you want the phone number appended to be in a single, standard format. The `phoneNum` variable contains a string built from groups 1, 3, 5, and 8 of the matched text ❷. (These groups are the area code, first three digits, last four digits, and extension.)

Step 4: Join the Matches into a String for the Clipboard

Now that you have the email addresses and phone numbers as a list of strings in `matches`, you want to put them on the clipboard. The `pyperclip.copy()` function takes only a single string value, not a list of strings, so you call the `join()` method on `matches`.

To make it easier to see that the program is working, let's print any matches you find to the terminal. And if no phone numbers or email addresses were found, the program should tell the user this.

Make your program look like the following:

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

--snip--
for groups in emailRegex.findall(text):
    matches.append(groups[0])

# Copy results to the clipboard.
if len(matches) > 0:
    pyperclip.copy('\n'.join(matches))
    print('Copied to clipboard:')
    print('\n'.join(matches))
else:
    print('No phone numbers or email addresses found.)
```

Running the Program

For an example, open your web browser to the No Starch Press contact page at <http://www.nostarch.com/contactus.htm>, press CTRL-A to select all the text on the page, and press CTRL-C to copy it to the clipboard. When you run this program, the output will look something like this:

Copied to clipboard:
800-420-7240
415-863-9900
415-863-9950
info@nostarch.com
media@nostarch.com
academic@nostarch.com
help@nostarch.com

Ideas for Similar Programs

Identifying patterns of text (and possibly substituting them with the `sub()` method) has many different potential applications.

- Find website URLs that begin with `http://` or `https://`.
- Clean up dates in different date formats (such as 3/14/2015, 03-14-2015, and 2015/3/14) by replacing them with dates in a single, standard format.
- Remove sensitive information such as Social Security or credit card numbers.
- Find common typos such as multiple spaces between words, accidentally accidentally repeated words, or multiple exclamation marks at the end of sentences. Those are annoying!!

Summary

While a computer can search for text quickly, it must be told precisely what to look for. Regular expressions allow you to specify the precise patterns of characters you are looking for. In fact, some word processing and spreadsheet applications provide find-and-replace features that allow you to search using regular expressions.

The `re` module that comes with Python lets you compile Regex objects. These values have several methods: `search()` to find a single match, `.findall()` to find all matching instances, and `sub()` to do a find-and-replace substitution of text.

There's a bit more to regular expression syntax than is described in this chapter. You can find out more in the official Python documentation at <http://docs.python.org/3/library/re.html>. The tutorial website <http://www.regular-expressions.info/> is also a useful resource.

Now that you have expertise manipulating and matching strings, it's time to dive into how to read from and write to files on your computer's hard drive.

Practice Questions

1. What is the function that creates Regex objects?
2. Why are raw strings often used when creating Regex objects?
3. What does the `search()` method return?
4. How do you get the actual strings that match the pattern from a `Match` object?
5. In the regex created from `r'(\d\d\d)-(\d\d\d-\d\d\d\d\d)`, what does group 0 cover? Group 1? Group 2?
6. Parentheses and periods have specific meanings in regular expression syntax. How would you specify that you want a regex to match actual parentheses and period characters?
7. The `.findall()` method returns a list of strings or a list of tuples of strings. What makes it return one or the other?
8. What does the `|` character signify in regular expressions?
9. What two things does the `?` character signify in regular expressions?
10. What is the difference between the `+` and `*` characters in regular expressions?
11. What is the difference between `{3}` and `{3,5}` in regular expressions?
12. What do the `\d`, `\w`, and `\s` shorthand character classes signify in regular expressions?
13. What do the `\D`, `\W`, and `\S` shorthand character classes signify in regular expressions?
14. How do you make a regular expression case-insensitive?
15. What does the `.` character normally match? What does it match if `re.DOTALL` is passed as the second argument to `re.compile()`?
16. What is the difference between `.*` and `.*??`?
17. What is the character class syntax to match all numbers and lowercase letters?
18. If `numRegex = re.compile(r'\d+')`, what will `numRegex.sub('X', '12 drummers, 11 pipers, five rings, 3 hens')` return?
19. What does passing `re.VERBOSE` as the second argument to `re.compile()` allow you to do?
20. How would you write a regex that matches a number with commas for every three digits? It must match the following:
 - `'42'`
 - `'1,234'`
 - `'6,368,745'`but not the following:
 - `'12,34,567'` (which has only two digits between the commas)
 - `'1234'` (which lacks commas)

21. How would you write a regex that matches the full name of someone whose last name is Nakamoto? You can assume that the first name that comes before it will always be one word that begins with a capital letter. The regex must match the following:

- 'Satoshi Nakamoto'
- 'Alice Nakamoto'
- 'RoboCop Nakamoto'

but not the following:

- 'satoshi Nakamoto' (where the first name is not capitalized)
- 'Mr. Nakamoto' (where the preceding word has a nonletter character)
- 'Nakamoto' (which has no first name)
- 'Satoshi nakamoto' (where Nakamoto is not capitalized)

22. How would you write a regex that matches a sentence where the first word is either *Alice*, *Bob*, or *Carol*; the second word is either *eats*, *pets*, or *throws*; the third word is *apples*, *cats*, or *baseballs*; and the sentence ends with a period? This regex should be case-insensitive. It must match the following:

- 'Alice eats apples.'
- 'Bob pets cats.'
- 'Carol throws baseballs.'
- 'Alice throws Apples.'
- 'BOB EATS CATS.'

but not the following:

- 'RoboCop eats apples.'
- 'ALICE THROWS FOOTBALLS.'
- 'Carol eats 7 cats.'

Practice Projects

For practice, write programs to do the following tasks.

Strong Password Detection

Write a function that uses regular expressions to make sure the password string it is passed is strong. A strong password is defined as one that is at least eight characters long, contains both uppercase and lowercase characters, and has at least one digit. You may need to test the string against multiple regex patterns to validate its strength.

Regex Version of strip()

Write a function that takes a string and does the same thing as the `strip()` string method. If no other arguments are passed other than the string to `strip`, then whitespace characters will be removed from the beginning and end of the string. Otherwise, the characters specified in the second argument to the function will be removed from the string.

8

READING AND WRITING FILES



Variables are a fine way to store data while your program is running, but if you want your data to persist even after your program has finished, you need to save it to a file. You can think of a file's contents as a single string value, potentially gigabytes in size. In this chapter, you will learn how to use Python to create, read, and save files on the hard drive.

Files and File Paths

A file has two key properties: a *filename* (usually written as one word) and a *path*. The path specifies the location of a file on the computer. For example, there is a file on my Windows 7 laptop with the filename *projects.docx* in the path *C:\Users\asweigart\Documents*. The part of the filename after the last period is called the file's *extension* and tells you a file's type. *project.docx* is a Word document, and *Users*, *asweigart*, and *Documents* all refer to *folders* (also

called *directories*). Folders can contain files and other folders. For example, *project.docx* is in the *Documents* folder, which is inside the *asweigart* folder, which is inside the *Users* folder. Figure 8-1 shows this folder organization.

The *C:* part of the path is the *root folder*, which contains all other folders. On Windows, the root folder is named *C:* and is also called the *C: drive*. On OS X and Linux, the root folder is */*. In this book, I'll be using the Windows-style root folder, *C:*. If you are entering the interactive shell examples on OS X or Linux, enter */* instead.

Additional *volumes*, such as a DVD drive or USB thumb drive, will appear differently on different operating systems. On Windows, they appear as new, lettered root drives, such as *D:* or *E:*. On OS X, they appear as new folders under the */Volumes* folder. On Linux, they appear as new folders under the */mnt* (“mount”) folder. Also note that while folder names and filenames are not case sensitive on Windows and OS X, they are case sensitive on Linux.



Figure 8-1: A file in a hierarchy of folders

Backslash on Windows and Forward Slash on OS X and Linux

On Windows, paths are written using backslashes (**) as the separator between folder names. OS X and Linux, however, use the forward slash (*/*) as their path separator. If you want your programs to work on all operating systems, you will have to write your Python scripts to handle both cases.

Fortunately, this is simple to do with the `os.path.join()` function. If you pass it the string values of individual file and folder names in your path, `os.path.join()` will return a string with a file path using the correct path separators. Enter the following into the interactive shell:

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
'usr\bin\spam'
```

I'm running these interactive shell examples on Windows, so `os.path.join('usr', 'bin', 'spam')` returned `'usr\bin\spam'`. (Notice that the backslashes are doubled because each backslash needs to be escaped by another backslash character.) If I had called this function on OS X or Linux, the string would have been `'usr/bin/spam'`.

The `os.path.join()` function is helpful if you need to create strings for filenames. These strings will be passed to several of the file-related functions introduced in this chapter. For example, the following example joins names from a list of filenames to the end of a folder's name:

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
```

```
print(os.path.join('C:\\\\Users\\\\asweigart', filename))
C:\\Users\\asweigart\\accounts.txt
C:\\Users\\asweigart\\details.csv
C:\\Users\\asweigart\\invite.docx
```

The Current Working Directory

Every program that runs on your computer has a *current working directory*, or *cwd*. Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory. You can get the current working directory as a string value with the `os.getcwd()` function and change it with `os.chdir()`. Enter the following into the interactive shell:

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

Here, the current working directory is set to *C:\Python34*, so the file-name *project.docx* refers to *C:\Python34\project.docx*. When we change the current working directory to *C:\Windows*, *project.docx* is interpreted as *C:\Windows\project.docx*.

Python will display an error if you try to change to a directory that does not exist.

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:\\ThisFolderDoesNotExist'
```

NOTE

While *folder* is the more modern name for *directory*, note that current working directory (or just working directory) is the standard term, not current working folder.

Absolute vs. Relative Paths

There are two ways to specify a file path.

- An *absolute path*, which always begins with the root folder
- A *relative path*, which is relative to the program's current working directory

There are also the *dot* (.) and *dot-dot* (..) folders. These are not real folders but special names that can be used in a path. A single period ("dot") for a folder name is shorthand for "this directory." Two periods ("dot-dot") means "the parent folder."

Figure 8-2 is an example of some folders and files. When the current working directory is set to *C:\bacon*, the relative paths for the other folders and files are set as they are in the figure.



Figure 8-2: The relative paths for folders and files in the working directory *C:\bacon*

The *.* at the start of a relative path is optional. For example, *.spam.txt* and *spam.txt* refer to the same file.

Creating New Folders with `os.makedirs()`

Your programs can create new folders (directories) with the `os.makedirs()` function. Enter the following into the interactive shell:

```
>>> import os  
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

This will create not just the *C:\delicious* folder but also a *walnut* folder inside *C:\delicious* and a *waffles* folder inside *C:\delicious\walnut*. That is, `os.makedirs()` will create any necessary intermediate folders in order to ensure that the full path exists. Figure 8-3 shows this hierarchy of folders.

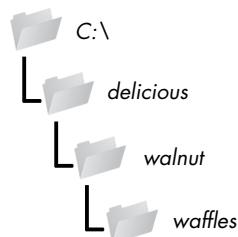


Figure 8-3: The result of `os.makedirs('C:\\delicious\\walnut\\waffles')`

The `os.path` Module

The `os.path` module contains many helpful functions related to filenames and file paths. For instance, you've already used `os.path.join()` to build paths in a way that will work on any operating system. Since `os.path` is a module inside the `os` module, you can import it by simply running `import os`. Whenever your programs need to work with files, folders, or file paths, you can refer to the short examples in this section. The full documentation for the `os.path` module is on the Python website at <http://docs.python.org/3/library/os.path.html>.

NOTE

Most of the examples that follow in this section will require the `os` module, so remember to import it at the beginning of any script you write and any time you restart IDLE. Otherwise, you'll get a `NameError: name 'os' is not defined` error message.

Handling Absolute and Relative Paths

The `os.path` module provides functions for returning the absolute path of a relative path and for checking whether a given path is an absolute path.

- Calling `os.path.abspath(path)` will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one.
- Calling `os.path.isabs(path)` will return `True` if the argument is an absolute path and `False` if it is a relative path.
- Calling `os.path.relpath(path, start)` will return a string of a relative path from the `start` path to `path`. If `start` is not provided, the current working directory is used as the start path.

Try these functions in the interactive shell:

```
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('.\\Scripts')
'C:\\Python34\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

Since `C:\\Python34` was the working directory when `os.path.abspath()` was called, the “single-dot” folder represents the absolute path `'C:\\Python34'`.

NOTE

Since your system probably has different files and folders on it than mine, you won't be able to follow every example in this chapter exactly. Still, try to follow along using folders that exist on your computer.

Enter the following calls to `os.path.relpath()` into the interactive shell:

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd()
'C:\\Python34'
```

Calling `os.path.dirname(path)` will return a string of everything that comes before the last slash in the path argument. Calling `os.path.basename(path)` will return a string of everything that comes after the last slash in the path argument. The dir name and base name of a path are outlined in Figure 8-4.



Figure 8-4: The base name follows the last slash in a path and is the same as the filename. The dir name is everything before the last slash.

For example, enter the following into the interactive shell:

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```

If you need a path's dir name and base name together, you can just call `os.path.split()` to get a tuple value with these two strings, like so:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
('C:\\Windows\\System32', 'calc.exe')
```

Notice that you could create the same tuple by calling `os.path.dirname()` and `os.path.basename()` and placing their return values in a tuple.

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))
('C:\\Windows\\System32', 'calc.exe')
```

But `os.path.split()` is a nice shortcut if you need both values.

Also, note that `os.path.split()` does *not* take a file path and return a list of strings of each folder. For that, use the `split()` string method and split on the string in `os.sep`. Recall from earlier that the `os.sep` variable is set to the correct folder-separating slash for the computer running the program.

For example, enter the following into the interactive shell:

```
>>> calcFilePath.split(os.path.sep)
['C:', 'Windows', 'System32', 'calc.exe']
```

On OS X and Linux systems, there will be a blank string at the start of the returned list:

```
>>> '/usr/bin'.split(os.path.sep)
 ['', 'usr', 'bin']
```

The `split()` string method will work to return a list of each part of the path. It will work on any operating system if you pass it `os.path.sep`.

Finding File Sizes and Folder Contents

Once you have ways of handling file paths, you can then start gathering information about specific files and folders. The `os.path` module provides functions for finding the size of a file in bytes and the files and folders inside a given folder.

- Calling `os.path.getsize(path)` will return the size in bytes of the file in the `path` argument.
- Calling `os.listdir(path)` will return a list of filename strings for each file in the `path` argument. (Note that this function is in the `os` module, not `os.path`.)

Here's what I get when I try these functions in the interactive shell:

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
776192
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--snip--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

As you can see, the `calc.exe` program on my computer is 776,192 bytes in size, and I have a lot of files in `C:\Windows\System32`. If I want to find the total size of all the files in this directory, I can use `os.path.getsize()` and `os.listdir()` together.

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
    totalSize = totalSize + os.path.getsize(os.path.join('C:\\Windows\\System32', filename))

>>> print(totalSize)
1117846456
```

As I loop over each filename in the `C:\Windows\System32` folder, the `totalSize` variable is incremented by the size of each file. Notice how when I call `os.path.getsize()`, I use `os.path.join()` to join the folder name with the current filename. The integer that `os.path.getsize()` returns is added to the value of `totalSize`. After looping through all the files, I print `totalSize` to see the total size of the `C:\Windows\System32` folder.

Checking Path Validity

Many Python functions will crash with an error if you supply them with a path that does not exist. The `os.path` module provides functions to check whether a given path exists and whether it is a file or folder.

- Calling `os.path.exists(path)` will return `True` if the file or folder referred to in the argument exists and will return `False` if it does not exist.
- Calling `os.path.isfile(path)` will return `True` if the path argument exists and is a file and will return `False` otherwise.
- Calling `os.path.isdir(path)` will return `True` if the path argument exists and is a folder and will return `False` otherwise.

Here's what I get when I try these functions in the interactive shell:

```
>>> os.path.exists('C:\\Windows')
True
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

You can determine whether there is a DVD or flash drive currently attached to the computer by checking for it with the `os.path.exists()` function. For instance, if I wanted to check for a flash drive with the volume named `D:\\` on my Windows computer, I could do that with the following:

```
>>> os.path.exists('D:\\')
False
```

Oops! It looks like I forgot to plug in my flash drive.

The File Reading/Writing Process

Once you are comfortable working with folders and relative paths, you'll be able to specify the location of files to read and write. The functions covered in the next few sections will apply to plaintext files. *Plaintext files*

contain only basic text characters and do not include font, size, or color information. Text files with the `.txt` extension or Python script files with the `.py` extension are examples of plaintext files. These can be opened with Windows's Notepad or OS X'sTextEdit application. Your programs can easily read the contents of plaintext files and treat them as an ordinary string value.

Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs. If you open a binary file in Notepad orTextEdit, it will look like scrambled nonsense, like in Figure 8-5.



Figure 8-5: The Windows `calc.exe` program opened in Notepad

Since every different type of binary file must be handled in its own way, this book will not go into reading and writing raw binary files directly. Fortunately, many modules make working with binary files easier—you will explore one of them, the `shelve` module, later in this chapter.

There are three steps to reading or writing files in Python.

1. Call the `open()` function to return a `File` object.
2. Call the `read()` or `write()` method on the `File` object.
3. Close the file by calling the `close()` method on the `File` object.

Opening Files with the `open()` Function

To open a file with the `open()` function, you pass it a string path indicating the file you want to open; it can be either an absolute or relative path. The `open()` function returns a `File` object.

Try it by creating a text file named `hello.txt` using Notepad orTextEdit. Type `Hello world!` as the content of this text file and save it in your user home folder. Then, if you're using Windows, enter the following into the interactive shell:

```
>>> helloFile = open('C:\\\\Users\\\\your_home_folder\\\\hello.txt')
```

If you're using OS X, enter the following into the interactive shell instead:

```
>>> helloFile = open('/Users/your_home_folder/hello.txt')
```

Make sure to replace *your_home_folder* with your computer username. For example, my username is *asweigart*, so I'd enter 'C:\\Users\\asweigart\\hello.txt' on Windows.

Both these commands will open the file in “reading plaintext” mode, or *read mode* for short. When a file is opened in read mode, Python lets you only read data from the file; you can’t write or modify it in any way. Read mode is the default mode for files you open in Python. But if you don’t want to rely on Python’s defaults, you can explicitly specify the mode by passing the string value ‘r’ as a second argument to `open()`. So `open('/Users/asweigart/hello.txt', 'r')` and `open('/Users/asweigart/hello.txt')` do the same thing.

The call to `open()` returns a `File` object. A `File` object represents a file on your computer; it is simply another type of value in Python, much like the lists and dictionaries you’re already familiar with. In the previous example, you stored the `File` object in the variable `helloFile`. Now, whenever you want to read from or write to the file, you can do so by calling methods on the `File` object in `helloFile`.

Reading the Contents of Files

Now that you have a `File` object, you can start reading from it. If you want to read the entire contents of a file as a string value, use the `File` object’s `read()` method. Let’s continue with the `hello.txt` `File` object you stored in `helloFile`. Enter the following into the interactive shell:

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello world!'
```

If you think of the contents of a file as a single large string value, the `read()` method returns the string that is stored in the file.

Alternatively, you can use the `readlines()` method to get a *list* of string values from the file, one string for each line of text. For example, create a file named `sonnet29.txt` in the same directory as `hello.txt` and write the following text in it:

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
```

Make sure to separate the four lines with line breaks. Then enter the following into the interactive shell:

```
>>> sonnetFile = open('sonnet29.txt')
>>> sonnetFile.readlines()
[When, in disgrace with fortune and men's eyes,\n', ' I all alone beweep my
outcast state,\n', And trouble deaf heaven with my bootless cries,\n', And
look upon myself and curse my fate,']
```

Note that each of the string values ends with a newline character, `\n`, except for the last line of the file. A list of strings is often easier to work with than a single large string value.

Writing to Files

Python allows you to write content to a file in a way similar to how the `print()` function “writes” strings to the screen. You can’t write to a file you’ve opened in read mode, though. Instead, you need to open it in “write plaintext” mode or “append plaintext” mode, or *write mode* and *append mode* for short.

Write mode will overwrite the existing file and start from scratch, just like when you overwrite a variable’s value with a new value. Pass `'w'` as the second argument to `open()` to open the file in write mode. Append mode, on the other hand, will append text to the end of the existing file. You can think of this as appending to a list in a variable, rather than overwriting the variable altogether. Pass `'a'` as the second argument to `open()` to open the file in append mode.

If the filename passed to `open()` does not exist, both write and append mode will create a new, blank file. After reading or writing a file, call the `close()` method before opening the file again.

Let’s put these concepts together. Enter the following into the interactive shell:

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

First, we open `bacon.txt` in write mode. Since there isn’t a `bacon.txt` yet, Python creates one. Calling `write()` on the opened file and passing `write()` the string argument `'Hello world! \n'` writes the string to the file and returns the number of characters written, including the newline. Then we close the file.

To add text to the existing contents of the file instead of replacing the string we just wrote, we open the file in append mode. We write `'Bacon is not a vegetable.'` to the file and close it. Finally, to print the file contents to the screen, we open the file in its default read mode, call `read()`, store the resulting File object in `content`, close the file, and print `content`.

Note that the `write()` method does not automatically add a newline character to the end of the string like the `print()` function does. You will have to add this character yourself.

Saving Variables with the `shelve` Module

You can save variables in your Python programs to binary shelf files using the `shelve` module. This way, your program can restore data to variables from the hard drive. The `shelve` module will let you add Save and Open features to your program. For example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run.

Enter the following into the interactive shell:

```
>>> import shelve
>>> shelfFile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
```

To read and write data using the `shelve` module, you first import `shelve`. Call `shelve.open()` and pass it a filename, and then store the returned shelf value in a variable. You can make changes to the shelf value as if it were a dictionary. When you're done, call `close()` on the shelf value. Here, our shelf value is stored in `shelfFile`. We create a list `cats` and write `shelfFile['cats'] = cats` to store the list in `shelfFile` as a value associated with the key '`cats`' (like in a dictionary). Then we call `close()` on `shelfFile`.

After running the previous code on Windows, you will see three new files in the current working directory: `mydata.bak`, `mydata.dat`, and `mydata.dir`. On OS X, only a single `mydata.db` file will be created.

These binary files contain the data you stored in your shelf. The format of these binary files is not important; you only need to know what the `shelve` module does, not how it does it. The module frees you from worrying about how to store your program's data to a file.

Your programs can use the `shelve` module to later reopen and retrieve the data from these shelf files. Shelf values don't have to be opened in read or write mode—they can do both once opened. Enter the following into the interactive shell:

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close()
```

Here, we open the shelf files to check that our data was stored correctly. Entering `shelfFile['cats']` returns the same list that we stored earlier, so we know that the list is correctly stored, and we call `close()`.

Just like dictionaries, shelf values have `keys()` and `values()` methods that will return list-like values of the keys and values in the shelf. Since these methods return list-like values instead of true lists, you should pass them to the `list()` function to get them in list form. Enter the following into the interactive shell:

```
>>> shelfFile = shelve.open('mydata')
>>> list(shelfFile.keys())
['cats']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelfFile.close()
```

Plaintext is useful for creating files that you'll read in a text editor such as Notepad orTextEdit, but if you want to save data from your Python programs, use the `shelve` module.

Saving Variables with the `pprint.pformat()` Function

Recall from “Pretty Printing” on page 111 that the `pprint pprint()` function will “pretty print” the contents of a list or dictionary to the screen, while the `pprint.pformat()` function will return this same text as a string instead of printing it. Not only is this string formatted to be easy to read, but it is also syntactically correct Python code. Say you have a dictionary stored in a variable and you want to save this variable and its contents for future use. Using `pprint.pformat()` will give you a string that you can write to `.py` file. This file will be your very own module that you can import whenever you want to use the variable stored in it.

For example, enter the following into the interactive shell:

```
>>> import pprint
>>> cats = [{"name": "Zophie", "desc": "chubby"}, {"name": "Pooka", "desc": "fluffy"}]
>>> pprint.pformat(cats)
"[{"desc": "chubby", "name": "Zophie"}, {"desc": "fluffy", "name": "Pooka"}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

Here, we import `pprint` to let us use `pprint.pformat()`. We have a list of dictionaries, stored in a variable `cats`. To keep the list in `cats` available even after we close the shell, we use `pprint.pformat()` to return it as a string. Once we have the data in `cats` as a string, it's easy to write the string to a file, which we'll call `myCats.py`.

The modules that an `import` statement imports are themselves just Python scripts. When the string from `pprint.pformat()` is saved to a `.py` file, the file is a module that can be imported just like any other.

And since Python scripts are themselves just text files with the `.py` file extension, your Python programs can even generate other Python programs. You can then import these files into scripts.

```
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

The benefit of creating a `.py` file (as opposed to saving variables with the `shelve` module) is that because it is a text file, the contents of the file can be read and modified by anyone with a simple text editor. For most applications, however, saving data using the `shelve` module is the preferred way to save variables to a file. Only basic data types such as integers, floats, strings, lists, and dictionaries can be written to a file as simple text. File objects, for example, cannot be encoded as text.

Project: Generating Random Quiz Files

Say you're a geography teacher with 35 students in your class and you want to give a pop quiz on US state capitals. Alas, your class has a few bad eggs in it, and you can't trust the students not to cheat. You'd like to randomize the order of questions so that each quiz is unique, making it impossible for anyone to crib answers from anyone else. Of course, doing this by hand would be a lengthy and boring affair. Fortunately, you know some Python.

Here is what the program does:

- Creates 35 different quizzes.
- Creates 50 multiple-choice questions for each quiz, in random order.
- Provides the correct answer and three random wrong answers for each question, in random order.
- Writes the quizzes to 35 text files.
- Writes the answer keys to 35 text files.

This means the code will need to do the following:

- Store the states and their capitals in a dictionary.
- Call `open()`, `write()`, and `close()` for the quiz and answer key text files.
- Use `random.shuffle()` to randomize the order of the questions and multiple-choice options.

Step 1: Store the Quiz Data in a Dictionary

The first step is to create a skeleton script and fill it with your quiz data. Create a file named *randomQuizGenerator.py*, and make it look like the following:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

❶ import random

# The quiz data. Keys are states and values are their capitals.
❷ capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix',
'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver',
'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee',
'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois':
'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas':
'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine':
'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston', 'Michigan':
'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson', 'Missouri':
'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln', 'Nevada':
'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton', 'New
Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina': 'Raleigh',
'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City',
'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence',
'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee':
'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont':
'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West
Virginia': 'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}
```

```
# Generate 35 quiz files.
❸ for quizNum in range(35):
    # TODO: Create the quiz and answer key files.

    # TODO: Write out the header for the quiz.

    # TODO: Shuffle the order of the states.

    # TODO: Loop through all 50 states, making a question for each.
```

Since this program will be randomly ordering the questions and answers, you'll need to import the `random` module ❶ to make use of its functions. The `capitals` variable ❷ contains a dictionary with US states as keys and their capitals as values. And since you want to create 35 quizzes, the code that actually generates the quiz and answer key files (marked with `TODO` comments for now) will go inside a `for` loop that loops 35 times ❸. (This number can be changed to generate any number of quiz files.)

Step 2: Create the Quiz File and Shuffle the Question Order

Now it's time to start filling in those TODOs.

The code in the loop will be repeated 35 times—once for each quiz—so you have to worry about only one quiz at a time within the loop. First you'll create the actual quiz file. It needs to have a unique filename and should also have some kind of standard header in it, with places for the student to fill in a name, date, and class period. Then you'll need to get a list of states in randomized order, which can be used later to create the questions and answers for the quiz.

Add the following lines of code to *randomQuizGenerator.py*:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--

# Generate 35 quiz files.
for quizNum in range(35):
    # Create the quiz and answer key files.
①     quizFile = open('capitalsquiz%s.txt' % (quizNum + 1), 'w')
②     answerKeyFile = open('capitalsquiz_answers%s.txt' % (quizNum + 1), 'w')

    # Write out the header for the quiz.
③     quizFile.write('Name:\n\nDate:\n\nPeriod:\n\n')
    quizFile.write(( ' ' * 20) + 'State Capitals Quiz (Form %s)' % (quizNum + 1))
    quizFile.write('\n\n')

    # Shuffle the order of the states.
    states = list(capitals.keys())
④     random.shuffle(states)

    # TODO: Loop through all 50 states, making a question for each.
```

The filenames for the quizzes will be *capitalsquiz<N>.txt*, where <N> is a unique number for the quiz that comes from *quizNum*, the for loop's counter. The answer key for *capitalsquiz<N>.txt* will be stored in a text file named *capitalsquiz_answers<N>.txt*. Each time through the loop, the %s placeholder in 'capitalsquiz%s.txt' and 'capitalsquiz_answers%s.txt' will be replaced by (quizNum + 1), so the first quiz and answer key created will be *capitalsquiz1.txt* and *capitalsquiz_answers1.txt*. These files will be created with calls to the *open()* function at ① and ②, with 'w' as the second argument to open them in write mode.

The *write()* statements at ③ create a quiz header for the student to fill out. Finally, a randomized list of US states is created with the help of the *random.shuffle()* function ④, which randomly reorders the values in any list that is passed to it.

Step 3: Create the Answer Options

Now you need to generate the answer options for each question, which will be multiple choice from A to D. You'll need to create another for loop—this one to generate the content for each of the 50 questions on the quiz. Then there will be a third for loop nested inside to generate the multiple-choice options for each question. Make your code look like the following:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--

# Loop through all 50 states, making a question for each.
for questionNum in range(50):

    # Get right and wrong answers.
❶    correctAnswer = capitals[states[questionNum]]
❷    wrongAnswers = list(capitals.values())
❸    del wrongAnswers[wrongAnswers.index(correctAnswer)]
❹    wrongAnswers = random.sample(wrongAnswers, 3)
❺    answerOptions = wrongAnswers + [correctAnswer]
❻    random.shuffle(answerOptions)

    # TODO: Write the question and answer options to the quiz file.

    # TODO: Write the answer key to a file.
```

The correct answer is easy to get—it's stored as a value in the `capitals` dictionary ❶. This loop will loop through the states in the shuffled states list, from `states[0]` to `states[49]`, find each state in `capitals`, and store that state's corresponding capital in `correctAnswer`.

The list of possible wrong answers is trickier. You can get it by duplicating *all* the values in the `capitals` dictionary ❷, deleting the correct answer ❸, and selecting three random values from this list ❹. The `random.sample()` function makes it easy to do this selection. Its first argument is the list you want to select from; the second argument is the number of values you want to select. The full list of answer options is the combination of these three wrong answers with the correct answers ❺. Finally, the answers need to be randomized ❻ so that the correct response isn't always choice D.

Step 4: Write Content to the Quiz and Answer Key Files

All that is left is to write the question to the quiz file and the answer to the answer key file. Make your code look like the following:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--
```

```
# Loop through all 50 states, making a question for each.  
for questionNum in range(50):  
    --snip--  
  
        # Write the question and the answer options to the quiz file.  
        quizFile.write('%s. What is the capital of %s?\n' % (questionNum + 1,  
            states[questionNum]))  
❶    for i in range(4):  
❷        quizFile.write('    %s. %s\n' % ('ABCD'[i], answerOptions[i]))  
        quizFile.write('\n')  
  
    # Write the answer key to a file.  
❸    answerKeyFile.write('%s. %s\n' % (questionNum + 1, 'ABCD'[  
        answerOptions.index(correctAnswer)]))  
    quizFile.close()  
    answerKeyFile.close()
```

A for loop that goes through integers 0 to 3 will write the answer options in the `answerOptions` list ❶. The expression `'ABCD'[i]` at ❷ treats the string `'ABCD'` as an array and will evaluate to `'A'`, `'B'`, `'C'`, and then `'D'` on each respective iteration through the loop.

In the final line ❸, the expression `answerOptions.index(correctAnswer)` will find the integer index of the correct answer in the randomly ordered answer options, and `'ABCD'[answerOptions.index(correctAnswer)]` will evaluate to the correct answer's letter to be written to the answer key file.

After you run the program, this is how your `capitalsquiz1.txt` file will look, though of course your questions and answer options may be different from those shown here, depending on the outcome of your `random.shuffle()` calls:

Name:

Date:

Period:

State Capitals Quiz (Form 1)

1. What is the capital of West Virginia?
 - A. Hartford
 - B. Santa Fe
 - C. Harrisburg
 - D. Charleston

2. What is the capital of Colorado?
 - A. Raleigh
 - B. Harrisburg
 - C. Denver
 - D. Lincoln

--snip--

The corresponding *capitalsquiz_answers1.txt* text file will look like this:

1. D
 2. C
 3. A
 4. C
- snip--
-

Project: Multiclipboard

Say you have the boring task of filling out many forms in a web page or software with several text fields. The clipboard saves you from typing the same text over and over again. But only one thing can be on the clipboard at a time. If you have several different pieces of text that you need to copy and paste, you have to keep highlighting and copying the same few things over and over again.

You can write a Python program to keep track of multiple pieces of text. This “multiclipboard” will be named *mcb.pyw* (since “mcb” is shorter to type than “multiclipboard”). The *.pyw* extension means that Python won’t show a Terminal window when it runs this program. (See Appendix B for more details.)

The program will save each piece of clipboard text under a keyword. For example, when you run `py mcb.pyw save spam`, the current contents of the clipboard will be saved with the keyword *spam*. This text can later be loaded to the clipboard again by running `py mcb.pyw spam`. And if the user forgets what keywords they have, they can run `py mcb.pyw list` to copy a list of all keywords to the clipboard.

Here’s what the program does:

- The command line argument for the keyword is checked.
- If the argument is `save`, then the clipboard contents are saved to the keyword.
- If the argument is `list`, then all the keywords are copied to the clipboard.
- Otherwise, the text for the keyword is copied to the keyboard.

This means the code will need to do the following:

- Read the command line arguments from `sys.argv`.
- Read and write to the clipboard.
- Save and load to a shelf file.

If you use Windows, you can easily run this script from the Run... window by creating a batch file named *mcb.bat* with the following content:

```
@pyw.exe C:\Python34\mcb.pyw %*
```

Step 1: Comments and Shelf Setup

Let's start by making a skeleton script with some comments and basic setup. Make your code look like the following:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
❶ # Usage: py.exe mcb.pyw save <keyword> - Saves clipboard to keyword.
#           py.exe mcb.pyw <keyword> - Loads keyword to clipboard.
#           py.exe mcb.pyw list - Loads all keywords to clipboard.

❷ import shelve, pyperclip, sys

❸ mcbShelf = shelve.open('mcb')

# TODO: Save clipboard content.

# TODO: List keywords and load content.

mcbShelf.close()
```

It's common practice to put general usage information in comments at the top of the file ❶. If you ever forget how to run your script, you can always look at these comments for a reminder. Then you import your modules ❷. Copying and pasting will require the `pyperclip` module, and reading the command line arguments will require the `sys` module. The `shelve` module will also come in handy: Whenever the user wants to save a new piece of clipboard text, you'll save it to a shelf file. Then, when the user wants to paste the text back to their clipboard, you'll open the shelf file and load it back into your program. The shelf file will be named with the prefix `mcb` ❸.

Step 2: Save Clipboard Content with a Keyword

The program does different things depending on whether the user wants to save text to a keyword, load text into the clipboard, or list all the existing keywords. Let's deal with that first case. Make your code look like the following:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--

# Save clipboard content.
❶ if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
❷     mcbShelf[sys.argv[2]] = pyperclip.paste()
❸ elif len(sys.argv) == 2:
    # TODO: List keywords and load content.

mcbShelf.close()
```

If the first command line argument (which will always be at index 1 of the `sys.argv` list) is 'save' ❶, the second command line argument is the keyword for the current content of the clipboard. The keyword will be used as the key for `mcbShelf`, and the value will be the text currently on the clipboard ❷.

If there is only one command line argument, you will assume it is either 'list' or a keyword to load content onto the clipboard. You will implement that code later. For now, just put a TODO comment there ❸.

Step 3: List Keywords and Load a Keyword's Content

Finally, let's implement the two remaining cases: The user wants to load clipboard text in from a keyword, or they want a list of all available keywords. Make your code look like the following:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--

# Save clipboard content.
if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
    mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
    # List keywords and load content.
❶    if sys.argv[1].lower() == 'list':
❷        pyperclip.copy(str(list(mcbShelf.keys())))
❸    elif sys.argv[1] in mcbShelf:
        pyperclip.copy(mcbShelf[sys.argv[1]])

mcbShelf.close()
```

If there is only one command line argument, first let's check whether it's 'list' ❶. If so, a string representation of the list of shelf keys will be copied to the clipboard ❷. The user can paste this list into an open text editor to read it.

Otherwise, you can assume the command line argument is a keyword. If this keyword exists in the `mcbShelf` shelf as a key, you can load the value onto the clipboard ❸.

And that's it! Launching this program has different steps depending on what operating system your computer uses. See Appendix B for details for your operating system.

Recall the password locker program you created in Chapter 6 that stored the passwords in a dictionary. Updating the passwords required changing the source code of the program. This isn't ideal because average users don't feel comfortable changing source code to update their software. Also, every time you modify the source code to a program, you run the risk of accidentally introducing new bugs. By storing the data for a program in a different place than the code, you can make your programs easier for others to use and more resistant to bugs.

Summary

Files are organized into folders (also called directories), and a path describes the location of a file. Every program running on your computer has a current working directory, which allows you to specify file paths relative to the current location instead of always typing the full (or absolute) path. The `os.path` module has many functions for manipulating file paths.

Your programs can also directly interact with the contents of text files. The `open()` function can open these files to read in their contents as one large string (with the `read()` method) or as a list of strings (with the `readlines()` method). The `open()` function can open files in write or append mode to create new text files or add to existing text files, respectively.

In previous chapters, you used the clipboard as a way of getting large amounts of text into a program, rather than typing it all in. Now you can have your programs read files directly from the hard drive, which is a big improvement, since files are much less volatile than the clipboard.

In the next chapter, you will learn how to handle the files themselves, by copying them, deleting them, renaming them, moving them, and more.

Practice Questions

1. What is a relative path relative to?
2. What does an absolute path start with?
3. What do the `os.getcwd()` and `os.chdir()` functions do?
4. What are the `.` and `..` folders?
5. In `C:\bacon\eggs\spam.txt`, which part is the dir name, and which part is the base name?
6. What are the three “mode” arguments that can be passed to the `open()` function?
7. What happens if an existing file is opened in write mode?
8. What is the difference between the `read()` and `readlines()` methods?
9. What data structure does a shelf value resemble?

Practice Projects

For practice, design and write the following programs.

Extending the Multiclipboard

Extend the multiclipboard program in this chapter so that it has a `delete <keyword>` command line argument that will delete a keyword from the shelf. Then add a `delete` command line argument that will delete *all* keywords.

Mad Libs

Create a Mad Libs program that reads in text files and lets the user add their own text anywhere the word *ADJECTIVE*, *NOUN*, *ADVERB*, or *VERB* appears in the text file. For example, a text file may look like this:

The *ADJECTIVE* panda walked to the *NOUN* and then *VERB*. A nearby *NOUN* was unaffected by these events.

The program would find these occurrences and prompt the user to replace them.

Enter an adjective:

silly

Enter a noun:

chandelier

Enter a verb:

screamed

Enter a noun:

pickup truck

The following text file would then be created:

The **silly** panda walked to the **chandelier** and then **screamed**. A nearby **pickup truck** was unaffected by these events.

The results should be printed to the screen and saved to a new text file.

Regex Search

Write a program that opens all *.txt* files in a folder and searches for any line that matches a user-supplied regular expression. The results should be printed to the screen.

9

ORGANIZING FILES



In the previous chapter, you learned how to create and write to new files in Python. Your programs can also organize preexisting files on the hard drive. Maybe you've had the experience of going through a folder full of dozens, hundreds, or even thousands of files and copying, renaming, moving, or compressing them all by hand. Or consider tasks such as these:

- Making copies of all PDF files (and *only* the PDF files) in every sub-folder of a folder
- Removing the leading zeros in the filenames for every file in a folder of hundreds of files named *spam001.txt*, *spam002.txt*, *spam003.txt*, and so on
- Compressing the contents of several folders into one ZIP file (which could be a simple backup system)

All this boring stuff is just begging to be automated in Python. By programming your computer to do these tasks, you can transform it into a quick-working file clerk who never makes mistakes.

As you begin working with files, you may find it helpful to be able to quickly see what the extension (*.txt*, *.pdf*, *.jpg*, and so on) of a file is. With OS X and Linux, your file browser most likely shows extensions automatically. With Windows, file extensions may be hidden by default. To show extensions, go to **Start ▶ Control Panel ▶ Appearance and Personalization ▶ Folder Options**. On the View tab, under Advanced Settings, uncheck the **Hide extensions for known file types** checkbox.

The `shutil` Module

The `shutil` (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs. To use the `shutil` functions, you will first need to use `import shutil`.

Copying Files and Folders

The `shutil` module provides functions for copying files, as well as entire folders.

Calling `shutil.copy(source, destination)` will copy the file at the path *source* to the folder at the path *destination*. (Both *source* and *destination* are strings.) If *destination* is a filename, it will be used as the new name of the copied file. This function returns a string of the path of the copied file.

Enter the following into the interactive shell to see how `shutil.copy()` works:

```
>>> import shutil, os
>>> os.chdir('C:\\')
❶ >>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
'C:\\delicious\\spam.txt'
❷ >>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
'C:\\delicious\\eggs2.txt'
```

The first `shutil.copy()` call copies the file at *C:\spam.txt* to the folder *C:\delicious*. The return value is the path of the newly copied file. Note that since a folder was specified as the destination ❶, the original *spam.txt* filename is used for the new, copied file's filename. The second `shutil.copy()` call ❷ also copies the file at *C:\eggs.txt* to the folder *C:\delicious* but gives the copied file the name *eggs2.txt*.

While `shutil.copy()` will copy a single file, `shutil.copytree()` will copy an entire folder and every folder and file contained in it. Calling `shutil.copytree(source, destination)` will copy the folder at the path *source*, along with all of its files and subfolders, to the folder at the path *destination*. The *source* and *destination* parameters are both strings. The function returns a string of the path of the copied folder.

Enter the following into the interactive shell:

```
>>> import shutil, os  
>>> os.chdir('C:\\\\')  
>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')  
'C:\\bacon_backup'
```

The `shutil.copytree()` call creates a new folder named `bacon_backup` with the same content as the original `bacon` folder. You have now safely backed up your precious, precious bacon.

Moving and Renaming Files and Folders

Calling `shutil.move(source, destination)` will move the file or folder at the path `source` to the path `destination` and will return a string of the absolute path of the new location.

If `destination` points to a folder, the `source` file gets moved into `destination` and keeps its current filename. For example, enter the following into the interactive shell:

```
>>> import shutil  
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')  
'C:\\eggs\\bacon.txt'
```

Assuming a folder named `eggs` already exists in the `C:\\`directory, this `shutil.move()` calls says, “Move `C:\\bacon.txt` into the folder `C:\\eggs`.”

If there had been a `bacon.txt` file already in `C:\\eggs`, it would have been overwritten. Since it’s easy to accidentally overwrite files in this way, you should take some care when using `move()`.

The `destination` path can also specify a filename. In the following example, the `source` file is moved *and* renamed.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\\\new_bacon.txt')  
'C:\\eggs\\\\new_bacon.txt'
```

This line says, “Move `C:\\bacon.txt` into the folder `C:\\eggs`, and while you’re at it, rename that `bacon.txt` file to `new_bacon.txt`.”

Both of the previous examples worked under the assumption that there was a folder `eggs` in the `C:\\`directory. But if there is no `eggs` folder, then `move()` will rename `bacon.txt` to a file named `eggs`.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')  
'C:\\eggs'
```

Here, `move()` can’t find a folder named `eggs` in the `C:\\`directory and so assumes that `destination` must be specifying a filename, not a folder. So the `bacon.txt` text file is renamed to `eggs` (a text file without the `.txt` file extension)—probably not what you wanted! This can be a tough-to-spot bug in

your programs since the `move()` call can happily do something that might be quite different from what you were expecting. This is yet another reason to be careful when using `move()`.

Finally, the folders that make up the destination must already exist, or else Python will throw an exception. Enter the following into the interactive shell:

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
Traceback (most recent call last):
  File "C:\Python34\lib\shutil.py", line 521, in move
    os.rename(src, real_dst)
FileNotFoundError: [WinError 3] The system cannot find the path specified:
'spam.txt' -> 'c:\\does_not_exist\\eggs\\ham'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
  File "C:\Python34\lib\shutil.py", line 533, in move
    copy2(src, real_dst)
  File "C:\Python34\lib\shutil.py", line 244, in copy2
    copyfile(src, dst, follow_symlinks=follow_symlinks)
  File "C:\Python34\lib\shutil.py", line 108, in copyfile
    with open(dst, 'wb') as fdst:
FileNotFoundException: [Errno 2] No such file or directory: 'c:\\does_not_exist\\eggs\\ham'
```

Python looks for *eggs* and *ham* inside the directory *does_not_exist*. It doesn't find the nonexistent directory, so it can't move *spam.txt* to the path you specified.

Permanently Deleting Files and Folders

You can delete a single file or a single empty folder with functions in the `os` module, whereas to delete a folder and all of its contents, you use the `shutil` module.

- Calling `os.unlink(path)` will delete the file at `path`.
- Calling `os.rmdir(path)` will delete the folder at `path`. This folder must be empty of any files or folders.
- Calling `shutil.rmtree(path)` will remove the folder at `path`, and all files and folders it contains will also be deleted.

Be careful when using these functions in your programs! It's often a good idea to first run your program with these calls commented out and with `print()` calls added to show the files that would be deleted. Here is

a Python program that was intended to delete files that have the `.txt` file extension but has a typo (highlighted in bold) that causes it to delete `.rxt` files instead:

```
import os
for filename in os.listdir():
    if filename.endswith('.rxt'):
        os.unlink(filename)
```

If you had any important files ending with `.rxt`, they would have been accidentally, permanently deleted. Instead, you should have first run the program like this:

```
import os
for filename in os.listdir():
    if filename.endswith('.rxt'):
        #os.unlink(filename)
        print(filename)
```

Now the `os.unlink()` call is commented, so Python ignores it. Instead, you will print the filename of the file that would have been deleted. Running this version of the program first will show you that you've accidentally told the program to delete `.rxt` files instead of `.txt` files.

Once you are certain the program works as intended, delete the `print(filename)` line and uncomment the `os.unlink(filename)` line. Then run the program again to actually delete the files.

Safe Deletes with the send2trash Module

Since Python's built-in `shutil.rmtree()` function irreversibly deletes files and folders, it can be dangerous to use. A much better way to delete files and folders is with the third-party `send2trash` module. You can install this module by running `pip install send2trash` from a Terminal window. (See Appendix A for a more in-depth explanation of how to install third-party modules.)

Using `send2trash` is much safer than Python's regular delete functions, because it will send folders and files to your computer's trash or recycle bin instead of permanently deleting them. If a bug in your program deletes something with `send2trash` you didn't intend to delete, you can later restore it from the recycle bin.

After you have installed `send2trash`, enter the following into the interactive shell:

```
>>> import send2trash
>>> baconFile = open('bacon.txt', 'a')  # creates the file
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> send2trash.send2trash('bacon.txt')
```

In general, you should always use the `send2trash.send2trash()` function to delete files and folders. But while sending files to the recycle bin lets you recover them later, it will not free up disk space like permanently deleting them does. If you want your program to free up disk space, use the `os` and `shutil` functions for deleting files and folders. Note that the `send2trash()` function can only send files to the recycle bin; it cannot pull files out of it.

Walking a Directory Tree

Say you want to rename every file in some folder and also every file in every subfolder of that folder. That is, you want to walk through the directory tree, touching each file as you go. Writing a program to do this could get tricky; fortunately, Python provides a function to handle this process for you.

Let's look at the `C:\delicious` folder with its contents, shown in Figure 9-1.



Figure 9-1: An example folder that contains three folders and four files

Here is an example program that uses the `os.walk()` function on the directory tree from Figure 9-1:

```
import os

for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('The current folder is ' + folderName)

    for subfolder in subfolders:
        print('SUBFOLDER OF ' + folderName + ': ' + subfolder)
```

```
for filename in filenames:  
    print('FILE INSIDE ' + folderName + ': ' + filename)  
  
print('')
```

The `os.walk()` function is passed a single string value: the path of a folder. You can use `os.walk()` in a `for` loop statement to walk a directory tree, much like how you can use the `range()` function to walk over a range of numbers. Unlike `range()`, the `os.walk()` function will return three values on each iteration through the loop:

1. A string of the current folder's name
2. A list of strings of the folders in the current folder
3. A list of strings of the files in the current folder

(By current folder, I mean the folder for the current iteration of the `for` loop. The current working directory of the program is *not* changed by `os.walk()`.)

Just like you can choose the variable name `i` in the code `for i in range(10):`, you can also choose the variable names for the three values listed earlier. I usually use the names `foldername`, `subfolders`, and `filenames`.

When you run this program, it will output the following:

```
The current folder is C:\delicious  
SUBFOLDER OF C:\delicious: cats  
SUBFOLDER OF C:\delicious: walnut  
FILE INSIDE C:\delicious: spam.txt
```

```
The current folder is C:\delicious\cats  
FILE INSIDE C:\delicious\cats: catnames.txt  
FILE INSIDE C:\delicious\cats: zophie.jpg
```

```
The current folder is C:\delicious\walnut  
SUBFOLDER OF C:\delicious\walnut: waffles
```

```
The current folder is C:\delicious\walnut\waffles  
FILE INSIDE C:\delicious\walnut\waffles: butter.txt.
```

Since `os.walk()` returns lists of strings for the `subfolder` and `filename` variables, you can use these lists in their own `for` loops. Replace the `print()` function calls with your own custom code. (Or if you don't need one or both of them, remove the `for` loops.)

Compressing Files with the `zipfile` Module

You may be familiar with ZIP files (with the `.zip` file extension), which can hold the compressed contents of many other files. Compressing a file reduces its size, which is useful when transferring it over the Internet. And

since a ZIP file can also contain multiple files and subfolders, it's a handy way to package several files into one. This single file, called an *archive file*, can then be, say, attached to an email.

Your Python programs can both create and open (or *extract*) ZIP files using functions in the `zipfile` module. Say you have a ZIP file named `example.zip` that has the contents shown in Figure 9-2.

You can download this ZIP file from <http://nostarch.com/automatestuff/> or just follow along using a ZIP file already on your computer.



Figure 9-2: The contents of `example.zip`

Reading ZIP Files

To read the contents of a ZIP file, first you must create a `ZipFile` object (note the capital letters `Z` and `F`). `ZipFile` objects are conceptually similar to the `File` objects you saw returned by the `open()` function in the previous chapter: They are values through which the program interacts with the file. To create a `ZipFile` object, call the `zipfile.ZipFile()` function, passing it a string of the `.zip` file's filename. Note that `zipfile` is the name of the Python module, and `ZipFile()` is the name of the function.

For example, enter the following into the interactive shell:

```
>>> import zipfile, os
>>> os.chdir('C:\\')      # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats\\', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
13908
>>> spamInfo.compress_size
3828
❶ >>> 'Compressed file is %sx smaller!' % (round(spamInfo.file_size / spamInfo
.compress_size, 2))
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

A `ZipFile` object has a `namelist()` method that returns a list of strings for all the files and folders contained in the ZIP file. These strings can be passed to the `getinfo()` `ZipFile` method to return a `ZipInfo` object about that particular file. `ZipInfo` objects have their own attributes, such as `file_size` and `compress_size` in bytes, which hold integers of the original file size and compressed file size, respectively. While a `ZipFile` object represents an entire archive file, a `ZipInfo` object holds useful information about a *single file* in the archive.

The command at ❶ calculates how efficiently `example.zip` is compressed by dividing the original file size by the compressed file size and prints this information using a string formatted with `%s`.

Extracting from ZIP Files

The `extractall()` method for `ZipFile` objects extracts all the files and folders from a ZIP file into the current working directory.

```
>>> import zipfile, os
>>> os.chdir('C:\\\\')      # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
❶ >>> exampleZip.extractall()
>>> exampleZip.close()
```

After running this code, the contents of `example.zip` will be extracted to `C:\`. Optionally, you can pass a folder name to `extractall()` to have it extract the files into a folder other than the current working directory. If the folder passed to the `extractall()` method does not exist, it will be created. For instance, if you replaced the call at ❶ with `exampleZip.extractall('C:\\\\delicious')`, the code would extract the files from `example.zip` into a newly created `C:\\delicious` folder.

The `extract()` method for `ZipFile` objects will extract a single file from the ZIP file. Continue the interactive shell example:

```
>>> exampleZip.extract('spam.txt')
'C:\\spam.txt'
>>> exampleZip.extract('spam.txt', 'C:\\some\\\\new\\\\folders')
'C:\\some\\\\new\\\\folders\\\\spam.txt'
>>> exampleZip.close()
```

The string you pass to `extract()` must match one of the strings in the list returned by `namelist()`. Optionally, you can pass a second argument to `extract()` to extract the file into a folder other than the current working directory. If this second argument is a folder that doesn't yet exist, Python will create the folder. The value that `extract()` returns is the absolute path to which the file was extracted.

Creating and Adding to ZIP Files

To create your own compressed ZIP files, you must open the `ZipFile` object in *write mode* by passing '`w`' as the second argument. (This is similar to opening a text file in write mode by passing '`w`' to the `open()` function.)

When you pass a path to the `write()` method of a `ZipFile` object, Python will compress the file at that path and add it into the ZIP file. The `write()` method's first argument is a string of the filename to add. The second argument is the *compression type* parameter, which tells the computer what algorithm it should use to compress the files; you can always just set this value to `zipfile.ZIP_DEFLATED`. (This specifies the *deflate* compression algorithm, which works well on all types of data.) Enter the following into the interactive shell:

```
>>> import zipfile
>>> newZip = zipfile.ZipFile('new.zip', 'w')
>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
>>> newZip.close()
```

This code will create a new ZIP file named *new.zip* that has the compressed contents of *spam.txt*.

Keep in mind that, just as with writing to files, write mode will erase all existing contents of a ZIP file. If you want to simply add files to an existing ZIP file, pass 'a' as the second argument to `zipfile.ZipFile()` to open the ZIP file in *append mode*.

Project: Renaming Files with American-Style Dates to European-Style Dates

Say your boss emails you thousands of files with American-style dates (MM-DD-YYYY) in their names and needs them renamed to European-style dates (DD-MM-YYYY). This boring task could take all day to do by hand! Let's write a program to do it instead.

Here's what the program does:

- It searches all the filenames in the current working directory for American-style dates.
- When one is found, it renames the file with the month and day swapped to make it European-style.

This means the code will need to do the following:

- Create a regex that can identify the text pattern of American-style dates.
- Call `os.listdir()` to find all the files in the working directory.
- Loop over each filename, using the regex to check whether it has a date.
- If it has a date, rename the file with `shutil.move()`.

For this project, open a new file editor window and save your code as *renameDates.py*.

Step 1: Create a Regex for American-Style Dates

The first part of the program will need to import the necessary modules and create a regex that can identify MM-DD-YYYY dates. The to-do comments will remind you what's left to write in this program. Typing them as `TODO` makes them easy to find using IDLE's CTRL-F find feature. Make your code look like the following:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.
```

```
❶ import shutil, os, re

    # Create a regex that matches files with the American date format.
❷ datePattern = re.compile(r"""\^(.*?)(\d{1,2})-(\d{1,2})-(\d{4})""") # all text before the date
                                                # one or two digits for the month
```

```

((0|1|2|3)?\d)-          # one or two digits for the day
((19|20)\d\d)            # four digits for the year
(.*)$                     # all text after the date
❸ """", re.VERBOSE)

# TODO: Loop over the files in the working directory.

# TODO: Skip files without a date.

# TODO: Get the different parts of the filename.

# TODO: Form the European-style filename.

# TODO: Get the full, absolute file paths.

# TODO: Rename the files.

```

From this chapter, you know the `shutil.move()` function can be used to rename files: Its arguments are the name of the file to rename and the new filename. Because this function exists in the `shutil` module, you must import that module ❶.

But before renaming the files, you need to identify which files you want to rename. Filenames with dates such as `spam4-4-1984.txt` and `01-03-2014eggs.zip` should be renamed, while filenames without dates such as `littlebrother.epub` can be ignored.

You can use a regular expression to identify this pattern. After importing the `re` module at the top, call `re.compile()` to create a Regex object ❷. Passing `re.VERBOSE` for the second argument ❸ will allow whitespace and comments in the regex string to make it more readable.

The regular expression string begins with `^(.*?)` to match any text at the beginning of the filename that might come before the date. The `((0|1)?\d)` group matches the month. The first digit can be either 0 or 1, so the regex matches 12 for December but also 02 for February. This digit is also optional so that the month can be 04 or 4 for April. The group for the day is `((0|1|2|3)?\d)` and follows similar logic; 3, 03, and 31 are all valid numbers for days. (Yes, this regex will accept some invalid dates such as 4-31-2014, 2-29-2013, and 0-15-2014. Dates have a lot of thorny special cases that can be easy to miss. But for simplicity, the regex in this program works well enough.)

While 1885 is a valid year, you can just look for years in the 20th or 21st century. This will keep your program from accidentally matching nondate filenames with a date-like format, such as `10-10-1000.txt`.

The `(.*?)$` part of the regex will match any text that comes after the date.

Step 2: Identify the Date Parts from the Filenames

Next, the program will have to loop over the list of filename strings returned from `os.listdir()` and match them against the regex. Any files that do not

have a date in them should be skipped. For filenames that have a date, the matched text will be stored in several variables. Fill in the first three TODOs in your program with the following code:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

--snip--

# Loop over the files in the working directory.
for amerFilename in os.listdir('.'):
    mo = datePattern.search(amerFilename)

    # Skip files without a date.
①    if mo == None:
②        continue

③    # Get the different parts of the filename.
    beforePart = mo.group(1)
    monthPart  = mo.group(2)
    dayPart    = mo.group(4)
    yearPart   = mo.group(6)
    afterPart  = mo.group(8)

--snip--
```

If the `Match` object returned from the `search()` method is `None` ①, then the filename in `amerFilename` does not match the regular expression. The `continue` statement ② will skip the rest of the loop and move on to the next filename.

Otherwise, the various strings matched in the regular expression groups are stored in variables named `beforePart`, `monthPart`, `dayPart`, `yearPart`, and `afterPart` ③. The strings in these variables will be used to form the European-style filename in the next step.

To keep the group numbers straight, try reading the regex from the beginning and count up each time you encounter an opening parenthesis. Without thinking about the code, just write an outline of the regular expression. This can help you visualize the groups. For example:

```
datePattern = re.compile(r"""\^(\1) # all text before the date
    (\2 (\3) )-          # one or two digits for the month
    (\4 (\5) )-          # one or two digits for the day
    (\6 (\7) )           # four digits for the year
    (\8)$                # all text after the date
    """", re.VERBOSE)
```

Here, the numbers **1** through **8** represent the groups in the regular expression you wrote. Making an outline of the regular expression, with just the parentheses and group numbers, can give you a clearer understanding of your regex before you move on with the rest of the program.

Step 3: Form the New Filename and Rename the Files

As the final step, concatenate the strings in the variables made in the previous step with the European-style date: The date comes before the month. Fill in the three remaining TODOs in your program with the following code:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

--snip--

❶ # Form the European-style filename.
❷ euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPart +
    afterPart

❸ # Get the full, absolute file paths.
absWorkingDir = os.path.abspath('.')
amerFilename = os.path.join(absWorkingDir, amerFilename)
euroFilename = os.path.join(absWorkingDir, euroFilename)

❹ # Rename the files.
❺ print('Renaming "%s" to "%s"...' % (amerFilename, euroFilename))
❻ #shutil.move(amerFilename, euroFilename) # uncomment after testing
```

Store the concatenated string in a variable named euroFilename ❶. Then, pass the original filename in amerFilename and the new euroFilename variable to the `shutil.move()` function to rename the file ❻.

This program has the `shutil.move()` call commented out and instead prints the filenames that will be renamed ❺. Running the program like this first can let you double-check that the files are renamed correctly. Then you can uncomment the `shutil.move()` call and run the program again to actually rename the files.

Ideas for Similar Programs

There are many other reasons why you might want to rename a large number of files.

- To add a prefix to the start of the filename, such as adding `spam_` to rename `eggs.txt` to `spam_eggs.txt`
- To change filenames with European-style dates to American-style dates
- To remove the zeros from files such as `spam0042.txt`

Project: Backing Up a Folder into a ZIP File

Say you’re working on a project whose files you keep in a folder named `C:\AlsPythonBook`. You’re worried about losing your work, so you’d like to create ZIP file “snapshots” of the entire folder. You’d like to keep different versions, so you want the ZIP file’s filename to increment each time it is made; for example, `AlsPythonBook_1.zip`, `AlsPythonBook_2.zip`,

AlsPythonBook_3.zip, and so on. You could do this by hand, but it is rather annoying, and you might accidentally misnumber the ZIP files' names. It would be much simpler to run a program that does this boring task for you.

For this project, open a new file editor window and save it as *backupToZip.py*.

Step 1: Figure Out the ZIP File's Name

The code for this program will be placed into a function named `backupToZip()`. This will make it easy to copy and paste the function into other Python programs that need this functionality. At the end of the program, the function will be called to perform the backup. Make your program look like this:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

❶ import zipfile, os

def backupToZip(folder):
    # Backup the entire contents of "folder" into a ZIP file.

    folder = os.path.abspath(folder)    # make sure folder is absolute

    # Figure out the filename this code should use based on
    # what files already exist.
❷    number = 1
❸    while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

❹    # TODO: Create the ZIP file.

    # TODO: Walk the entire folder tree and compress the files in each folder.
    print('Done.')

backupToZip('C:\\\\delicious')
```

Do the basics first: Add the shebang (`#!`) line, describe what the program does, and import the `zipfile` and `os` modules ❶.

Define a `backupToZip()` function that takes just one parameter, `folder`. This parameter is a string path to the folder whose contents should be backed up. The function will determine what filename to use for the ZIP file it will create; then the function will create the file, walk the `folder` folder, and add each of the subfolders and files to the ZIP file. Write TODO comments for these steps in the source code to remind yourself to do them later ❷.

The first part, naming the ZIP file, uses the base name of the absolute path of `folder`. If the folder being backed up is *C:\delicious*, the ZIP file's name should be *delicious_N.zip*, where *N*=1 is the first time you run the program, *N*=2 is the second time, and so on.

You can determine what N should be by checking whether *delicious_1.zip* already exists, then checking whether *delicious_2.zip* already exists, and so on. Use a variable named `number` for N ❷, and keep incrementing it inside the loop that calls `os.path.exists()` to check whether the file exists ❸. The first nonexistent filename found will cause the loop to break, since it will have found the filename of the new zip.

Step 2: Create the New ZIP File

Next let's create the ZIP file. Make your program look like the following:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--
while True:
    zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
    if not os.path.exists(zipFilename):
        break
    number = number + 1

    # Create the ZIP file.
    print('Creating %s...' % (zipFilename))
❶    backupZip = zipfile.ZipFile(zipFilename, 'w')

    # TODO: Walk the entire folder tree and compress the files in each folder.
    print('Done.')

backupToZip('C:\\delicious')
```

Now that the new ZIP file's name is stored in the `zipFilename` variable, you can call `zipfile.ZipFile()` to actually create the ZIP file ❶. Be sure to pass '`w`' as the second argument so that the ZIP file is opened in write mode.

Step 3: Walk the Directory Tree and Add to the ZIP File

Now you need to use the `os.walk()` function to do the work of listing every file in the folder and its subfolders. Make your program look like the following:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--

    # Walk the entire folder tree and compress the files in each folder.
❶    for foldername, subfolders, filenames in os.walk(folder):
        print('Adding files in %s...' % (foldername))
        # Add the current folder to the ZIP file.
❷        backupZip.write(foldername)
```

```
# Add all the files in this folder to the ZIP file.  
❸ for filename in filenames:  
    newBase = os.path.basename(folder) + '_'  
    if filename.startswith(newBase) and filename.endswith('.zip')  
        continue # don't backup the backup ZIP files  
    backupZip.write(os.path.join(foldername, filename))  
backupZip.close()  
print('Done.')  
  
backupToZip('C:\\delicious')
```

You can use `os.walk()` in a `for` loop ❶, and on each iteration it will return the iteration's current folder name, the subfolders in that folder, and the filenames in that folder.

In the `for` loop, the folder is added to the ZIP file ❷. The nested `for` loop can go through each filename in the `filenames` list ❸. Each of these is added to the ZIP file, except for previously made backup ZIPs.

When you run this program, it will produce output that will look something like this:

```
Creating delicious_1.zip...  
Adding files in C:\\delicious...  
Adding files in C:\\delicious\\cats...  
Adding files in C:\\delicious\\waffles...  
Adding files in C:\\delicious\\walnut...  
Adding files in C:\\delicious\\walnut\\waffles...  
Done.
```

The second time you run it, it will put all the files in `C:\\delicious` into a ZIP file named `delicious_2.zip`, and so on.

Ideas for Similar Programs

You can walk a directory tree and add files to compressed ZIP archives in several other programs. For example, you can write programs that do the following:

- Walk a directory tree and archive just files with certain extensions, such as `.txt` or `.py`, and nothing else
- Walk a directory tree and archive every file except the `.txt` and `.py` ones
- Find the folder in a directory tree that has the greatest number of files or the folder that uses the most disk space

Summary

Even if you are an experienced computer user, you probably handle files manually with the mouse and keyboard. Modern file explorers make it easy to work with a few files. But sometimes you'll need to perform a task that would take hours using your computer's file explorer.

The `os` and `shutil` modules offer functions for copying, moving, renaming, and deleting files. When deleting files, you might want to use the `send2trash` module to move files to the recycle bin or trash rather than permanently deleting them. And when writing programs that handle files, it's a good idea to comment out the code that does the actual copy/move/rename/delete and add a `print()` call instead so you can run the program and verify exactly what it will do.

Often you will need to perform these operations not only on files in one folder but also on every folder in that folder, every folder in those folders, and so on. The `os.walk()` function handles this trek across the folders for you so that you can concentrate on what your program needs to do with the files in them.

The `zipfile` module gives you a way of compressing and extracting files in `.zip` archives through Python. Combined with the file-handling functions of `os` and `shutil`, `zipfile` makes it easy to package up several files from anywhere on your hard drive. These `.zip` files are much easier to upload to websites or send as email attachments than many separate files.

Previous chapters of this book have provided source code for you to copy. But when you write your own programs, they probably won't come out perfectly the first time. The next chapter focuses on some Python modules that will help you analyze and debug your programs so that you can quickly get them working correctly.

Practice Questions

1. What is the difference between `shutil.copy()` and `shutil.copytree()`?
2. What function is used to rename files?
3. What is the difference between the delete functions in the `send2trash` and `shutil` modules?
4. `ZipFile` objects have a `close()` method just like `File` objects' `close()` method. What `ZipFile` method is equivalent to `File` objects' `open()` method?

Practice Projects

For practice, write programs to do the following tasks.

Selective Copy

Write a program that walks through a folder tree and searches for files with a certain file extension (such as `.pdf` or `.jpg`). Copy these files from whatever location they are in to a new folder.

Deleting Unneeded Files

It's not uncommon for a few unneeded but humongous files or folders to take up the bulk of the space on your hard drive. If you're trying to free up

room on your computer, you'll get the most bang for your buck by deleting the most massive of the unwanted files. But first you have to find them.

Write a program that walks through a folder tree and searches for exceptionally large files or folders—say, ones that have a file size of more than 100MB. (Remember, to get a file's size, you can use `os.path.getsize()` from the `os` module.) Print these files with their absolute path to the screen.

Filling in the Gaps

Write a program that finds all files with a given prefix, such as `spam001.txt`, `spam002.txt`, and so on, in a single folder and locates any gaps in the numbering (such as if there is a `spam001.txt` and `spam003.txt` but no `spam002.txt`). Have the program rename all the later files to close this gap.

As an added challenge, write another program that can insert gaps into numbered files so that a new file can be added.

10

DEBUGGING



Now that you know enough to write more complicated programs, you may start finding not-so-simple bugs in them. This chapter covers some tools and techniques for finding the root cause of bugs in your program to help you fix bugs faster and with less effort.

To paraphrase an old joke among programmers, “Writing code accounts for 90 percent of programming. Debugging code accounts for the other 90 percent.”

Your computer will do only what you tell it to do; it won’t read your mind and do what you *intended* it to do. Even professional programmers create bugs all the time, so don’t feel discouraged if your program has a problem.

Fortunately, there are a few tools and techniques to identify what exactly your code is doing and where it’s going wrong. First, you will look at logging and assertions, two features that can help you detect bugs early. In general, the earlier you catch bugs, the easier they will be to fix.

Second, you will look at how to use the debugger. The debugger is a feature of IDLE that executes a program one instruction at a time, giving you a chance to inspect the values in variables while your code runs, and track how the values change over the course of your program. This is much slower than running the program at full speed, but it is helpful to see the actual values in a program while it runs, rather than deducing what the values might be from the source code.

Raising Exceptions

Python raises an exception whenever it tries to execute invalid code. In Chapter 3, you read about how to handle Python’s exceptions with `try` and `except` statements so that your program can recover from exceptions that you anticipated. But you can also raise your own exceptions in your code. Raising an exception is a way of saying, “Stop running the code in this function and move the program execution to the `except` statement.”

Exceptions are raised with a `raise` statement. In code, a `raise` statement consists of the following:

- The `raise` keyword
- A call to the `Exception()` function
- A string with a helpful error message passed to the `Exception()` function

For example, enter the following into the interactive shell:

```
>>> raise Exception('This is the error message.')
Traceback (most recent call last):
  File "<pyshell#191>", line 1, in <module>
    raise Exception('This is the error message.')
Exception: This is the error message.
```

If there are no `try` and `except` statements covering the `raise` statement that raised the exception, the program simply crashes and displays the exception’s error message.

Often it’s the code that calls the function, not the function itself, that knows how to handle an exception. So you will commonly see a `raise` statement inside a function and the `try` and `except` statements in the code calling the function. For example, open a new file editor window, enter the following code, and save the program as `boxPrint.py`:

```
def boxPrint(symbol, width, height):
    if len(symbol) != 1:
        raise Exception('Symbol must be a single character string.')
    if width <= 2:
        raise Exception('Width must be greater than 2.')
    if height <= 2:
        raise Exception('Height must be greater than 2.')  
①  
②  
③
```

```
print(symbol * width)
for i in range(height - 2):
    print(symbol + (' ' * (width - 2)) + symbol)
print(symbol * width)

for sym, w, h in (('*', 4, 4), ('0', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
    try:
        boxPrint(sym, w, h)
❸ except Exception as err:
❹     print('An exception happened: ' + str(err))
```

Here we've defined a `boxPrint()` function that takes a character, a width, and a height, and uses the character to make a little picture of a box with that width and height. This box shape is printed to the console.

Say we want the character to be a single character, and the width and height to be greater than 2. We add if statements to raise exceptions if these requirements aren't satisfied. Later, when we call `boxPrint()` with various arguments, our `try/except` will handle invalid arguments.

This program uses the `except Exception as err` form of the `except` statement ❸. If an `Exception` object is returned from `boxPrint()` ❶❷❹, this `except` statement will store it in a variable named `err`. The `Exception` object can then be converted to a string by passing it to `str()` to produce a user-friendly error message ❹. When you run this `boxPrint.py`, the output will look like this:

```
*****
* *
* *
*****
000000000000000000000000
0          0
0          0
0          0
000000000000000000000000
An exception happened: Width must be greater than 2.
An exception happened: Symbol must be a single character string.
```

Using the `try` and `except` statements, you can handle errors more gracefully instead of letting the entire program crash.

Getting the Traceback as a String

When Python encounters an error, it produces a treasure trove of error information called the *traceback*. The traceback includes the error message, the line number of the line that caused the error, and the sequence of the function calls that led to the error. This sequence of calls is called the *call stack*.

Open a new file editor window in IDLE, enter the following program, and save it as `errorExample.py`:

```
def spam():
    bacon()
```

```
def bacon():
    raise Exception('This is the error message.')

spam()
```

When you run *errorExample.py*, the output will look like this:

```
Traceback (most recent call last):
  File "errorExample.py", line 7, in <module>
    spam()
  File "errorExample.py", line 2, in spam
    bacon()
  File "errorExample.py", line 5, in bacon
    raise Exception('This is the error message.')
Exception: This is the error message.
```

From the traceback, you can see that the error happened on line 5, in the `bacon()` function. This particular call to `bacon()` came from line 2, in the `spam()` function, which in turn was called on line 7. In programs where functions can be called from multiple places, the call stack can help you determine which call led to the error.

The traceback is displayed by Python whenever a raised exception goes unhandled. But you can also obtain it as a string by calling `traceback.format_exc()`. This function is useful if you want the information from an exception's traceback but also want an `except` statement to gracefully handle the exception. You will need to import Python's `traceback` module before calling this function.

For example, instead of crashing your program right when an exception occurs, you can write the traceback information to a log file and keep your program running. You can look at the log file later, when you're ready to debug your program. Enter the following into the interactive shell:

```
>>> import traceback
>>> try:
...     raise Exception('This is the error message.')
... except:
...     errorFile = open('errorInfo.txt', 'w')
...     errorFile.write(traceback.format_exc())
...     errorFile.close()
...     print('The traceback info was written to errorInfo.txt.')
```

```
116
The traceback info was written to errorInfo.txt.
```

The 116 is the return value from the `write()` method, since 116 characters were written to the file. The traceback text was written to *errorInfo.txt*.

```
Traceback (most recent call last):
  File "<pyshell#28>", line 2, in <module>
Exception: This is the error message.
```

Assertions

An *assertion* is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by `assert` statements. If the sanity check fails, then an `AssertionError` exception is raised. In code, an `assert` statement consists of the following:

- The `assert` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A comma
- A string to display when the condition is `False`

For example, enter the following into the interactive shell:

```
>>> podBayDoorStatus = 'open'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
>>> podBayDoorStatus = 'I'm sorry, Dave. I'm afraid I can't do that.'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
AssertionError: The pod bay doors need to be "open".
```

Here we've set `podBayDoorStatus` to `'open'`, so from now on, we fully expect the value of this variable to be `'open'`. In a program that uses this variable, we might have written a lot of code under the assumption that the value is `'open'`—code that depends on its being `'open'` in order to work as we expect. So we add an assertion to make sure we're right to assume `podBayDoorStatus` is `'open'`. Here, we include the message `'The pod bay doors need to be "open".'` so it'll be easy to see what's wrong if the assertion fails.

Later, say we make the obvious mistake of assigning `podBayDoorStatus` another value, but don't notice it among many lines of code. The assertion catches this mistake and clearly tells us what's wrong.

In plain English, an `assert` statement says, “I assert that this condition holds true, and if not, there is a bug somewhere in the program.” Unlike exceptions, your code should *not* handle `assert` statements with `try` and `except`; if an `assert` fails, your program *should* crash. By failing fast like this, you shorten the time between the original cause of the bug and when you first notice the bug. This will reduce the amount of code you will have to check before finding the code that's causing the bug.

Assertions are for programmer errors, not user errors. For errors that can be recovered from (such as a file not being found or the user entering invalid data), raise an exception instead of detecting it with an `assert` statement.

Using an Assertion in a Traffic Light Simulation

Say you're building a traffic light simulation program. The data structure representing the stoplights at an intersection is a dictionary with

keys 'ns' and 'ew', for the stoplights facing north-south and east-west, respectively. The values at these keys will be one of the strings 'green', 'yellow', or 'red'. The code would look something like this:

```
market_2nd = {'ns': 'green', 'ew': 'red'}
mission_16th = {'ns': 'red', 'ew': 'green'}
```

These two variables will be for the intersections of Market Street and 2nd Street, and Mission Street and 16th Street. To start the project, you want to write a `switchLights()` function, which will take an intersection dictionary as an argument and switch the lights.

At first, you might think that `switchLights()` should simply switch each light to the next color in the sequence: Any 'green' values should change to 'yellow', 'yellow' values should change to 'red', and 'red' values should change to 'green'. The code to implement this idea might look like this:

```
def switchLights(stolight):
    for key in stolight.keys():
        if stolight[key] == 'green':
            stolight[key] = 'yellow'
        elif stolight[key] == 'yellow':
            stolight[key] = 'red'
        elif stolight[key] == 'red':
            stolight[key] = 'green'

switchLights(market_2nd)
```

You may already see the problem with this code, but let's pretend you wrote the rest of the simulation code, thousands of lines long, without noticing it. When you finally do run the simulation, the program doesn't crash—but your virtual cars do!

Since you've already written the rest of the program, you have no idea where the bug could be. Maybe it's in the code simulating the cars or in the code simulating the virtual drivers. It could take hours to trace the bug back to the `switchLights()` function.

But if while writing `switchLights()` you had added an assertion to check that *at least one of the lights is always red*, you might have included the following at the bottom of the function:

```
assert 'red' in stolight.values(), 'Neither light is red! ' + str(stolight)
```

With this assertion in place, your program would crash with this error message:

```
Traceback (most recent call last):
  File "carSim.py", line 14, in <module>
    switchLights(market_2nd)
  File "carSim.py", line 13, in switchLights
    assert 'red' in stolight.values(), 'Neither light is red! ' + str(stolight)
❶ AssertionError: Neither light is red! {'ns': 'yellow', 'ew': 'green'}
```

The important line here is the `AssertionError` ❶. While your program crashing is not ideal, it immediately points out that a sanity check failed: Neither direction of traffic has a red light, meaning that traffic could be going both ways. By failing fast early in the program's execution, you can save yourself a lot of future debugging effort.

Disabling Assertions

Assertions can be disabled by passing the `-O` option when running Python. This is good for when you have finished writing and testing your program and don't want it to be slowed down by performing sanity checks (although most of the time `assert` statements do not cause a noticeable speed difference). Assertions are for development, not the final product. By the time you hand off your program to someone else to run, it should be free of bugs and not require the sanity checks. See Appendix B for details about how to launch your probably-not-insane programs with the `-O` option.

Logging

If you've ever put a `print()` statement in your code to output some variable's value while your program is running, you've used a form of *logging* to debug your code. Logging is a great way to understand what's happening in your program and in what order its happening. Python's `logging` module makes it easy to create a record of custom messages that you write. These log messages will describe when the program execution has reached the logging function call and list any variables you have specified at that point in time. On the other hand, a missing log message indicates a part of the code was skipped and never executed.

Using the logging Module

To enable the `logging` module to display log messages on your screen as your program runs, copy the following to the top of your program (but under the `#!/ python` shebang line):

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s
- %(message)s')
```

You don't need to worry too much about how this works, but basically, when Python logs an event, it creates a `LogRecord` object that holds information about that event. The `logging` module's `basicConfig()` function lets you specify what details about the `LogRecord` object you want to see and how you want those details displayed.

Say you wrote a function to calculate the *factorial* of a number. In mathematics, factorial 4 is $1 \times 2 \times 3 \times 4$, or 24. Factorial 7 is $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7$, or 5,040. Open a new file editor window and enter the following code. It has a bug in it, but you will also enter several log messages to help yourself figure out what is going wrong. Save the program as `factorialLog.py`.

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s
- %(message)s')
logging.debug('Start of program')

def factorial(n):
    logging.debug('Start of factorial(%s%%)' % (n))
    total = 1
    for i in range(n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)' % (n))
    return total

print(factorial(5))
logging.debug('End of program')
```

Here, we use the `logging.debug()` function when we want to print log information. This `debug()` function will call `basicConfig()`, and a line of information will be printed. This information will be in the format we specified in `basicConfig()` and will include the messages we passed to `debug()`. The `print(factorial(5))` call is part of the original program, so the result is displayed even if logging messages are disabled.

The output of this program looks like this:

```
2015-05-23 16:20:12,664 - DEBUG - Start of program
2015-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2015-05-23 16:20:12,684 - DEBUG - End of program
```

The `factorial()` function is returning 0 as the factorial of 5, which isn't right. The `for` loop should be multiplying the value in `total` by the numbers from 1 to 5. But the log messages displayed by `logging.debug()` show that the `i` variable is starting at 0 instead of 1. Since zero times anything is zero, the rest of the iterations also have the wrong value for `total`. Logging messages provide a trail of breadcrumbs that can help you figure out when things started to go wrong.

Change the `for i in range(n + 1):` line to `for i in range(1, n + 1):`, and run the program again. The output will look like this:

```
2015-05-23 17:13:40,650 - DEBUG - Start of program
2015-05-23 17:13:40,651 - DEBUG - Start of factorial(5)
2015-05-23 17:13:40,651 - DEBUG - i is 1, total is 1
2015-05-23 17:13:40,654 - DEBUG - i is 2, total is 2
2015-05-23 17:13:40,656 - DEBUG - i is 3, total is 6
```

```
2015-05-23 17:13:40,659 - DEBUG - i is 4, total is 24
2015-05-23 17:13:40,661 - DEBUG - i is 5, total is 120
2015-05-23 17:13:40,661 - DEBUG - End of factorial(5)
120
2015-05-23 17:13:40,666 - DEBUG - End of program
```

The `factorial(5)` call correctly returns 120. The log messages showed what was going on inside the loop, which led straight to the bug.

You can see that the `logging.debug()` calls printed out not just the strings passed to them but also a timestamp and the word `DEBUG`.

Don't Debug with `print()`

Typing `import logging` and `logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')` is somewhat unwieldy. You may want to use `print()` calls instead, but don't give in to this temptation! Once you're done debugging, you'll end up spending a lot of time removing `print()` calls from your code for each log message. You might even accidentally remove some `print()` calls that were being used for nonlog messages. The nice thing about log messages is that you're free to fill your program with as many as you like, and you can always disable them later by adding a single `logging.disable(logging.CRITICAL)` call. Unlike `print()`, the `logging` module makes it easy to switch between showing and hiding log messages.

Log messages are intended for the programmer, not the user. The user won't care about the contents of some dictionary value you need to see to help with debugging; use a log message for something like that. For messages that the user will want to see, like *File not found* or *Invalid input, please enter a number*, you should use a `print()` call. You don't want to deprive the user of useful information after you've disabled log messages.

Logging Levels

Logging levels provide a way to categorize your log messages by importance. There are five logging levels, described in Table 10-1 from least to most important. Messages can be logged at each level using a different logging function.

Table 10-1: Logging Levels in Python

Level	Logging Function	Description
DEBUG	<code>logging.debug()</code>	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	<code>logging.info()</code>	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING	<code>logging.warning()</code>	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.

(continued)

Table 10-1 (continued)

Level	Logging Function	Description
ERROR	logging.error()	Used to record an error that caused the program to fail to do something.
CRITICAL	logging.critical()	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Your logging message is passed as a string to these functions. The logging levels are suggestions. Ultimately, it is up to you to decide which category your log message falls into. Enter the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s - %(message)s')
>>> logging.debug('Some debugging details.')
2015-05-18 19:04:26,901 - DEBUG - Some debugging details.
>>> logging.info('The logging module is working.')
2015-05-18 19:04:35,569 - INFO - The logging module is working.
>>> logging.warning('An error message is about to be logged.')
2015-05-18 19:04:56,843 - WARNING - An error message is about to be logged.
>>> logging.error('An error has occurred.')
2015-05-18 19:05:07,737 - ERROR - An error has occurred.
>>> logging.critical('The program is unable to recover!')
2015-05-18 19:05:45,794 - CRITICAL - The program is unable to recover!
```

The benefit of logging levels is that you can change what priority of logging message you want to see. Passing logging.DEBUG to the basicConfig() function's level keyword argument will show messages from all the logging levels (DEBUG being the lowest level). But after developing your program some more, you may be interested only in errors. In that case, you can set basicConfig()'s level argument to logging.ERROR. This will show only ERROR and CRITICAL messages and skip the DEBUG, INFO, and WARNING messages.

Disabling Logging

After you've debugged your program, you probably don't want all these log messages cluttering the screen. The logging.disable() function disables these so that you don't have to go into your program and remove all the logging calls by hand. You simply pass logging.disable() a logging level, and it will suppress all log messages at that level or lower. So if you want to disable logging entirely, just add logging.disable(logging.CRITICAL) to your program. For example, enter the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s -
%(levelname)s - %(message)s')
```

```
>>> logging.critical('Critical error! Critical error!')
2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!
>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Critical error! Critical error!')
>>> logging.error('Error! Error!')
```

Since `logging.disable()` will disable all messages after it, you will probably want to add it near the `import logging` line of code in your program. This way, you can easily find it to comment out or uncomment that call to enable or disable logging messages as needed.

Logging to a File

Instead of displaying the log messages to the screen, you can write them to a text file. The `logging.basicConfig()` function takes a `filename` keyword argument, like so:

```
import logging
logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')
```

The log messages will be saved to `myProgramLog.txt`. While logging messages are helpful, they can clutter your screen and make it hard to read the program’s output. Writing the logging messages to a file will keep your screen clear and store the messages so you can read them after running the program. You can open this text file in any text editor, such as Notepad orTextEdit.

IDLE’s Debugger

The *debugger* is a feature of IDLE that allows you to execute your program one line at a time. The debugger will run a single line of code and then wait for you to tell it to continue. By running your program “under the debugger” like this, you can take as much time as you want to examine the values in the variables at any given point during the program’s lifetime. This is a valuable tool for tracking down bugs.

To enable IDLE’s debugger, click **Debug ▶ Debugger** in the interactive shell window. This will bring up the Debug Control window, which looks like Figure 10-1.

When the Debug Control window appears, select all four of the **Stack**, **Locals**, **Source**, and **Globals** checkboxes so that the window shows the full set of debug information. While the Debug Control window is displayed, any time you run a program from the file editor, the debugger will pause execution before the first instruction and display the following:

- The line of code that is about to be executed
- A list of all local variables and their values
- A list of all global variables and their values



Figure 10-1: The Debug Control window

You'll notice that in the list of global variables there are several variables you haven't defined, such as `_builtins_`, `_doc_`, `_file_`, and so on. These are variables that Python automatically sets whenever it runs a program. The meaning of these variables is beyond the scope of this book, and you can comfortably ignore them.

The program will stay paused until you press one of the five buttons in the Debug Control window: Go, Step, Over, Out, or Quit.

Go

Clicking the Go button will cause the program to execute normally until it terminates or reaches a *breakpoint*. (Breakpoints are described later in this chapter.) If you are done debugging and want the program to continue normally, click the Go button.

Step

Clicking the Step button will cause the debugger to execute the next line of code and then pause again. The Debug Control window's list of global and local variables will be updated if their values change. If the next line of code is a function call, the debugger will "step into" that function and jump to the first line of code of that function.

Over

Clicking the Over button will execute the next line of code, similar to the Step button. However, if the next line of code is a function call, the Over button will "step over" the code in the function. The function's code will be executed at full speed, and the debugger will pause as soon as the function call returns. For example, if the next line of code is a `print()` call, you don't

really care about code inside the built-in `print()` function; you just want the string you pass it printed to the screen. For this reason, using the Over button is more common than the Step button.

Out

Clicking the Out button will cause the debugger to execute lines of code at full speed until it returns from the current function. If you have stepped into a function call with the Step button and now simply want to keep executing instructions until you get back out, click the **Out** button to “step out” of the current function call.

Quit

If you want to stop debugging entirely and not bother to continue executing the rest of the program, click the **Quit** button. The Quit button will immediately terminate the program. If you want to run your program normally again, select **Debug ▶ Debugger** again to disable the debugger.

Debugging a Number Adding Program

Open a new file editor window and enter the following code:

```
print('Enter the first number to add:')
first = input()
print('Enter the second number to add:')
second = input()
print('Enter the third number to add:')
third = input()
print('The sum is ' + first + second + third)
```

Save it as `buggyAddingProgram.py` and run it first without the debugger enabled. The program will output something like this:

```
Enter the first number to add:
5
Enter the second number to add:
3
Enter the third number to add:
42
The sum is 5342
```

The program hasn’t crashed, but the sum is obviously wrong. Let’s enable the Debug Control window and run it again, this time under the debugger.

When you press F5 or select **Run ▶ Run Module** (with **Debug ▶ Debugger** enabled and all four checkboxes on the Debug Control window checked), the program starts in a paused state on line 1. The debugger will always pause on the line of code it is about to execute. The Debug Control window will look like Figure 10-2.



Figure 10-2: The Debug Control window when the program first starts under the debugger

Click the **Over** button once to execute the first `print()` call. You should use Over instead of Step here, since you don't want to step into the code for the `print()` function. The Debug Control window will update to line 2, and line 2 in the file editor window will be highlighted, as shown in Figure 10-3. This shows you where the program execution currently is.

The screenshot shows two windows side-by-side:

- File Editor Window:** Shows the code for `buggyAddingProgram.py`. The line `first = input()` is highlighted in yellow, indicating it is the current line of execution.
- Debug Control Window:** Shows the updated state:
 - Toolbar:** Go, Step, Over, Out, Quit, Stack, Source, Locals, Globals.
 - Status Bar:** buggyAddingProgram.py:2: <module>()
 - Text Area:** 'bdb'.run(), line 431: exec(cmd, globals, locals)

> _main_<module>(), line 2: first = input()
 - Locals:** None
 - Globals:**
 - _builtins_ <module 'builtins' (built-in)>
 - _doc_ None
 - _file_ 'C:\\\\buggyAddingProgram.py'
 - _loader_ <class '_frozen_importlib.BuiltinImporter'>
 - _name_ '_main_'
 - _package_ None
 - _spec_ None

Figure 10-3: The Debug Control window after clicking Over

Click **Over** again to execute the `input()` function call, and the buttons in the Debug Control window will disable themselves while IDLE waits for you to type something for the `input()` call into the interactive shell window. Enter `5` and press Return. The Debug Control window buttons will be reenabled.

Keep clicking **Over**, entering `3` and `42` as the next two numbers, until the debugger is on line 7, the final `print()` call in the program. The Debug Control window should look like Figure 10-4. You can see in the Globals section that the first, second, and third variables are set to string values '`5`', '`3`', and '`42`' instead of integer values `5`, `3`, and `42`. When the last line is executed, these strings are concatenated instead of added together, causing the bug.



Figure 10-4: The Debug Control window on the last line.
The variables are set to strings, causing the bug.

Stepping through the program with the debugger is helpful but can also be slow. Often you'll want the program to run normally until it reaches a certain line of code. You can configure the debugger to do this with breakpoints.

Breakpoints

A *breakpoint* can be set on a specific line of code and forces the debugger to pause whenever the program execution reaches that line. Open a new file editor window and enter the following program, which simulates flipping a coin 1,000 times. Save it as `coinFlip.py`.

```
import random
heads = 0
for i in range(1, 1001):
❶    if random.randint(0, 1) == 1:
        heads = heads + 1
    if i == 500:
❷        print('Halfway done!')
print('Heads came up ' + str(heads) + ' times.')
```

The `random.randint(0, 1)` call ❶ will return 0 half of the time and 1 the other half of the time. This can be used to simulate a 50/50 coin flip where 1 represents heads. When you run this program without the debugger, it quickly outputs something like the following:

```
Halfway done!
Heads came up 490 times.
```

If you ran this program under the debugger, you would have to click the Over button thousands of times before the program terminated. If you were interested in the value of `heads` at the halfway point of the program's execution, when 500 of 1000 coin flips have been completed, you could instead just set a breakpoint on the line `print('Halfway done!')` ❷. To set a breakpoint, right-click the line in the file editor and select **Set Breakpoint**, as shown in Figure 10-5.



Figure 10-5: Setting a breakpoint

You don't want to set a breakpoint on the `if` statement line, since the `if` statement is executed on every single iteration through the loop. By setting the breakpoint on the code in the `if` statement, the debugger breaks only when the execution enters the `if` clause.

The line with the breakpoint will be highlighted in yellow in the file editor. When you run the program under the debugger, it will start in a paused state at the first line, as usual. But if you click Go, the program will run at full speed until it reaches the line with the breakpoint set on it. You can then click Go, Over, Step, or Out to continue as normal.

If you want to remove a breakpoint, right-click the line in the file editor and select **Clear Breakpoint** from the menu. The yellow highlighting will go away, and the debugger will not break on that line in the future.

Summary

Assertions, exceptions, logging, and the debugger are all valuable tools to find and prevent bugs in your program. Assertions with the Python assert statement are a good way to implement “sanity checks” that give you an early warning when a necessary condition doesn’t hold true. Assertions are only for errors that the program shouldn’t try to recover from and should fail fast. Otherwise, you should raise an exception.

An exception can be caught and handled by the try and except statements. The logging module is a good way to look into your code while it’s running and is much more convenient to use than the print() function because of its different logging levels and ability to log to a text file.

The debugger lets you step through your program one line at a time. Alternatively, you can run your program at normal speed and have the debugger pause execution whenever it reaches a line with a breakpoint set. Using the debugger, you can see the state of any variable’s value at any point during the program’s lifetime.

These debugging tools and techniques will help you write programs that work. Accidentally introducing bugs into your code is a fact of life, no matter how many years of coding experience you have.

Practice Questions

1. Write an assert statement that triggers an `AssertionError` if the variable `spam` is an integer less than 10.
2. Write an assert statement that triggers an `AssertionError` if the variables `eggs` and `bacon` contain strings that are the same as each other, even if their cases are different (that is, `'hello'` and `'hello'` are considered the same, and `'goodbye'` and `'GOODbye'` are also considered the same).
3. Write an assert statement that *always* triggers an `AssertionError`.
4. What are the two lines that your program must have in order to be able to call `logging.debug()`?
5. What are the two lines that your program must have in order to have `logging.debug()` send a logging message to a file named `programLog.txt`?
6. What are the five logging levels?
7. What line of code can you add to disable all logging messages in your program?
8. Why is using logging messages better than using `print()` to display the same message?
9. What are the differences between the Step, Over, and Out buttons in the Debug Control window?

10. After you click Go in the Debug Control window, when will the debugger stop?
11. What is a breakpoint?
12. How do you set a breakpoint on a line of code in IDLE?

Practice Project

For practice, write a program that does the following.

Debugging Coin Toss

The following program is meant to be a simple coin toss guessing game. The player gets two guesses (it's an easy game). However, the program has several bugs in it. Run through the program a few times to find the bugs that keep the program from working correctly.

```
import random
guess = ''
while guess not in ('heads', 'tails'):
    print('Guess the coin toss! Enter heads or tails:')
    guess = input()
toss = random.randint(0, 1) # 0 is tails, 1 is heads
if toss == guess:
    print('You got it!')
else:
    print('Nope! Guess again!')
guesss = input()
if toss == guesss:
    print('You got it!')
else:
    print('Nope. You are really bad at this game.')
```

11

WEB SCRAPING



In those rare, terrifying moments when I'm without Wi-Fi, I realize just how much of what I do on the computer is really what I do on the Internet. Out of sheer habit I'll find myself trying to check email, read friends' Twitter feeds, or answer the question, "Did Kurtwood Smith have any major roles before he was in the original 1987 *RoboCop*?"¹

Since so much work on a computer involves going on the Internet, it'd be great if your programs could get online. *Web scraping* is the term for using a program to download and process content from the Web. For example, Google runs many web scraping programs to index web pages for its search engine. In this chapter, you will learn about several modules that make it easy to scrape web pages in Python.

1. The answer is no.

webbrowser Comes with Python and opens a browser to a specific page.

Requests Downloads files and web pages from the Internet.

Beautiful Soup Parses HTML, the format that web pages are written in.

Selenium Launches and controls a web browser. Selenium is able to fill in forms and simulate mouse clicks in this browser.

Project: *mapIt.py* with the `webbrowser` Module

The `webbrowser` module's `open()` function can launch a new browser to a specified URL. Enter the following into the interactive shell:

```
>>> import webbrowser  
>>> webbrowser.open('http://inventwithpython.com/')
```

A web browser tab will open to the URL `http://inventwithpython.com/`. This is about the only thing the `webbrowser` module can do. Even so, the `open()` function does make some interesting things possible. For example, it's tedious to copy a street address to the clipboard and bring up a map of it on Google Maps. You could take a few steps out of this task by writing a simple script to automatically launch the map in your browser using the contents of your clipboard. This way, you only have to copy the address to a clipboard and run the script, and the map will be loaded for you.

This is what your program does:

- Gets a street address from the command line arguments or clipboard.
- Opens the web browser to the Google Maps page for the address.

This means your code will need to do the following:

- Read the command line arguments from `sys.argv`.
- Read the clipboard contents.
- Call the `webbrowser.open()` function to open the web browser.

Open a new file editor window and save it as *mapIt.py*.

Step 1: Figure Out the URL

Based on the instructions in Appendix B, set up *mapIt.py* so that when you run it from the command line, like so . . .

```
C:\> mapit 870 Valencia St, San Francisco, CA 94110
```

. . . the script will use the command line arguments instead of the clipboard. If there are no command line arguments, then the program will know to use the contents of the clipboard.

First you need to figure out what URL to use for a given street address. When you load `http://maps.google.com/` in the browser and search for an address, the URL in the address bar looks something like this: `https://www.google.com/maps/place/870+Valencia+St/@37.7590311,-122.4215096,17z/data=!3m1!4b1!4m2!3m1!1s0x808f7e3dad07a37:0xc86b0b2bb93b73d8.`

The address is in the URL, but there's a lot of additional text there as well. Websites often add extra data to URLs to help track visitors or customize sites. But if you try just going to `https://www.google.com/maps/place/870+Valencia+St+San+Francisco+CA/`, you'll find that it still brings up the correct page. So your program can be set to open a web browser to '`https://www.google.com/maps/place/your_address_string`' (where `your_address_string` is the address you want to map).

Step 2: Handle the Command Line Arguments

Make your code look like this:

```
#! python3
# mapIt.py - Launches a map in the browser using an address from the
# command line or clipboard.

import webbrowser, sys
if len(sys.argv) > 1:
    # Get address from command line.
    address = ' '.join(sys.argv[1:])

# TODO: Get address from clipboard.
```

After the program's `#!` shebang line, you need to import the `webbrowser` module for launching the browser and import the `sys` module for reading the potential command line arguments. The `sys.argv` variable stores a list of the program's filename and command line arguments. If this list has more than just the filename in it, then `len(sys.argv)` evaluates to an integer greater than 1, meaning that command line arguments have indeed been provided.

Command line arguments are usually separated by spaces, but in this case, you want to interpret all of the arguments as a single string. Since `sys.argv` is a list of strings, you can pass it to the `join()` method, which returns a single string value. You don't want the program name in this string, so instead of `sys.argv`, you should pass `sys.argv[1:]` to chop off the first element of the array. The final string that this expression evaluates to is stored in the `address` variable.

If you run the program by entering this into the command line . . .

```
mapit 870 Valencia St, San Francisco, CA 94110
```

. . . the `sys.argv` variable will contain this list value:

```
['mapIt.py', '870', 'Valencia', 'St', ' ', 'San', 'Francisco', ' ', 'CA', '94110']
```

The `address` variable will contain the string '`870 Valencia St, San Francisco, CA 94110`'.

Step 3: Handle the Clipboard Content and Launch the Browser

Make your code look like the following:

```
#! python3
# mapIt.py - Launches a map in the browser using an address from the
# command line or clipboard.

import webbrowser, sys, pyperclip
if len(sys.argv) > 1:
    # Get address from command line.
    address = ' '.join(sys.argv[1:])
else:
    # Get address from clipboard.
    address = pyperclip.paste()

webbrowser.open('https://www.google.com/maps/place/' + address)
```

If there are no command line arguments, the program will assume the address is stored on the clipboard. You can get the clipboard content with `pyperclip.paste()` and store it in a variable named `address`. Finally, to launch a web browser with the Google Maps URL, call `webbrowser.open()`.

While some of the programs you write will perform huge tasks that save you hours, it can be just as satisfying to use a program that conveniently saves you a few seconds each time you perform a common task, such as getting a map of an address. Table 11-1 compares the steps needed to display a map with and without *mapIt.py*.

Table 11-1: Getting a Map with and Without *mapIt.py*

Manually getting a map	Using <i>mapIt.py</i>
Highlight the address.	Highlight the address.
Copy the address.	Copy the address.
Open the web browser.	Run <i>mapIt.py</i> .
Go to http://maps.google.com/ .	
Click the address text field.	
Paste the address.	
Press ENTER.	

See how *mapIt.py* makes this task less tedious?

Ideas for Similar Programs

As long as you have a URL, the `webbrowser` module lets users cut out the step of opening the browser and directing themselves to a website. Other programs could use this functionality to do the following:

- Open all links on a page in separate browser tabs.
- Open the browser to the URL for your local weather.
- Open several social network sites that you regularly check.

Downloading Files from the Web with the requests Module

The `requests` module lets you easily download files from the Web without having to worry about complicated issues such as network errors, connection problems, and data compression. The `requests` module doesn't come with Python, so you'll have to install it first. From the command line, run `pip install requests`. (Appendix A has additional details on how to install third-party modules.)

The `requests` module was written because Python's `urllib2` module is too complicated to use. In fact, take a permanent marker and black out this entire paragraph. Forget I ever mentioned `urllib2`. If you need to download things from the Web, just use the `requests` module.

Next, do a simple test to make sure the `requests` module installed itself correctly. Enter the following into the interactive shell:

```
>>> import requests
```

If no error messages show up, then the `requests` module has been successfully installed.

Downloading a Web Page with the `requests.get()` Function

The `requests.get()` function takes a string of a URL to download. By calling `type()` on `requests.get()`'s return value, you can see that it returns a `Response` object, which contains the response that the web server gave for your request. I'll explain the `Response` object in more detail later, but for now, enter the following into the interactive shell while your computer is connected to the Internet:

```
>>> import requests
❶ >>> res = requests.get('http://www.gutenberg.org/cache/epub/1112/pg1112.txt')
>>> type(res)
<class 'requests.models.Response'>
❷ >>> res.status_code == requests.codes.ok
True
>>> len(res.text)
178981
>>> print(res.text[:250])
The Project Gutenberg EBook of Romeo and Juliet, by William Shakespeare
```

This eBook is for the use of anyone anywhere at no cost and with almost no restrictions whatsoever. You may copy it, give it away or re-use it under the terms of the Projec

The URL goes to a text web page for the entire play of *Romeo and Juliet*, provided by Project Gutenberg ❶. You can tell that the request for this web page succeeded by checking the `status_code` attribute of the `Response` object.

If it is equal to the value of `requests.codes.ok`, then everything went fine **❷**. (Incidentally, the status code for “OK” in the HTTP protocol is 200. You may already be familiar with the 404 status code for “Not Found.”)

If the request succeeded, the downloaded web page is stored as a string in the `Response` object’s `text` variable. This variable holds a large string of the entire page; the call to `len(res.text)` shows you that it is more than 178,000 characters long. Finally, calling `print(res.text[:250])` displays only the first 250 characters.

Checking for Errors

As you’ve seen, the `Response` object has a `status_code` attribute that can be checked against `requests.codes.ok` to see whether the download succeeded. A simpler way to check for success is to call the `raise_for_status()` method on the `Response` object. This will raise an exception if there was an error downloading the file and will do nothing if the download succeeded. Enter the following into the interactive shell:

```
>>> res = requests.get('http://inventwithpython.com/page_that_does_not_exist')
>>> res.raise_for_status()
Traceback (most recent call last):
  File "<pyshell#138>", line 1, in <module>
    res.raise_for_status()
  File "C:\Python34\lib\site-packages\requests\models.py", line 773, in raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 404 Client Error: Not Found
```

The `raise_for_status()` method is a good way to ensure that a program halts if a bad download occurs. This is a good thing: You want your program to stop as soon as some unexpected error happens. If a failed download *isn’t* a deal breaker for your program, you can wrap the `raise_for_status()` line with `try` and `except` statements to handle this error case without crashing.

```
import requests
res = requests.get('http://inventwithpython.com/page_that_does_not_exist')
try:
    res.raise_for_status()
except Exception as exc:
    print('There was a problem: %s' % (exc))
```

This `raise_for_status()` method call causes the program to output the following:

```
There was a problem: 404 Client Error: Not Found
```

Always call `raise_for_status()` after calling `requests.get()`. You want to be sure that the download has actually worked before your program continues.

Saving Downloaded Files to the Hard Drive

From here, you can save the web page to a file on your hard drive with the standard `open()` function and `write()` method. There are some slight differences, though. First, you must open the file in *write binary* mode by passing the string '`wb`' as the second argument to `open()`. Even if the page is in plain-text (such as the *Romeo and Juliet* text you downloaded earlier), you need to write binary data instead of text data in order to maintain the *Unicode encoding* of the text.

UNICODE ENCODINGS

Unicode encodings are beyond the scope of this book, but you can learn more about them from these web pages:

- Joel on Software: The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!): <http://www.joelonsoftware.com/articles/Unicode.html>
- Pragmatic Unicode: <http://nedbatchelder.com/text/unipain.html>

To write the web page to a file, you can use a `for` loop with the `Response` object's `iter_content()` method.

```
>>> import requests
>>> res = requests.get('http://www.gutenberg.org/cache/epub/1112/pg1112.txt')
>>> res.raise_for_status()
>>> playFile = open('RomeoAndJuliet.txt', 'wb')
>>> for chunk in res.iter_content(100000):
    playFile.write(chunk)

100000
78981
>>> playFile.close()
```

The `iter_content()` method returns “chunks” of the content on each iteration through the loop. Each chunk is of the `bytes` data type, and you get to specify how many bytes each chunk will contain. One hundred thousand bytes is generally a good size, so pass `100000` as the argument to `iter_content()`.

The file *RomeoAndJuliet.txt* will now exist in the current working directory. Note that while the filename on the website was *pg1112.txt*, the file on your hard drive has a different filename. The `requests` module simply handles downloading the contents of web pages. Once the page is downloaded, it is simply data in your program. Even if you were to lose your Internet connection after downloading the web page, all the page data would still be on your computer.

The `write()` method returns the number of bytes written to the file. In the previous example, there were 100,000 bytes in the first chunk, and the remaining part of the file needed only 78,981 bytes.

To review, here's the complete process for downloading and saving a file:

1. Call `requests.get()` to download the file.
2. Call `open()` with '`wb`' to create a new file in write binary mode.
3. Loop over the `Response` object's `iter_content()` method.
4. Call `write()` on each iteration to write the content to the file.
5. Call `close()` to close the file.

That's all there is to the `requests` module! The `for` loop and `iter_content()` stuff may seem complicated compared to the `open()/write()/close()` workflow you've been using to write text files, but it's to ensure that the `requests` module doesn't eat up too much memory even if you download massive files. You can learn about the `requests` module's other features from <http://requests.readthedocs.org/>.

HTML

Before you pick apart web pages, you'll learn some HTML basics. You'll also see how to access your web browser's powerful developer tools, which will make scraping information from the Web much easier.

Resources for Learning HTML

Hypertext Markup Language (HTML) is the format that web pages are written in. This chapter assumes you have some basic experience with HTML, but if you need a beginner tutorial, I suggest one of the following sites:

- <http://htmldog.com/guides/html/beginner/>
- <http://www.codecademy.com/tracks/web/>
- <https://developer.mozilla.org/en-US/learn/html/>

A Quick Refresher

In case it's been a while since you've looked at any HTML, here's a quick overview of the basics. An HTML file is a plaintext file with the `.html` file extension. The text in these files is surrounded by *tags*, which are words enclosed in angle brackets. The tags tell the browser how to format the web page. A starting tag and closing tag can enclose some text to form an *element*. The *text* (or *inner HTML*) is the content between the starting and closing tags. For example, the following HTML will display *Hello world!* in the browser, with *Hello* in bold:

```
<strong>Hello</strong> world!
```

This HTML will look like Figure 11-1 in a browser.



Figure 11-1: Hello world! rendered in the browser

The opening `` tag says that the enclosed text will appear in bold. The closing `` tags tells the browser where the end of the bold text is.

There are many different tags in HTML. Some of these tags have extra properties in the form of *attributes* within the angle brackets. For example, the `<a>` tag encloses text that should be a link. The URL that the text links to is determined by the `href` attribute. Here's an example:

```
All's free <a href="http://inventwithpython.com">Python books</a>.
```

This HTML will look like Figure 11-2 in a browser.



Figure 11-2: The link rendered in the browser

Some elements have an `id` attribute that is used to uniquely identify the element in the page. You will often instruct your programs to seek out an element by its `id` attribute, so figuring out an element's `id` attribute using the browser's developer tools is a common task in writing web scraping programs.

Viewing the Source HTML of a Web Page

You'll need to look at the HTML source of the web pages that your programs will work with. To do this, right-click (or CTRL-click on OS X) any web page in your web browser, and select **View Source** or **View page source** to see the HTML text of the page (see Figure 11-3). This is the text your browser actually receives. The browser knows how to display, or *render*, the web page from this HTML.



Figure 11-3: Viewing the source of a web page

I highly recommend viewing the source HTML of some of your favorite sites. It's fine if you don't fully understand what you are seeing when you look at the source. You won't need HTML mastery to write simple web scraping programs—after all, you won't be writing your own websites. You just need enough knowledge to pick out data from an existing site.

Opening Your Browser's Developer Tools

In addition to viewing a web page's source, you can look through a page's HTML using your browser's developer tools. In Chrome and Internet Explorer for Windows, the developer tools are already installed, and you can press F12 to make them appear (see Figure 11-4). Pressing F12 again will make the developer tools disappear. In Chrome, you can also bring up the developer tools by selecting View ▶ Developer ▶ Developer Tools. In OS X, pressing ⌘-OPTION-I will open Chrome's Developer Tools.



Figure 11-4: The Developer Tools window in the Chrome browser

In Firefox, you can bring up the Web Developer Tools Inspector by pressing CTRL-SHIFT-C on Windows and Linux or by pressing ⌘-OPTION-C on OS X. The layout is almost identical to Chrome's developer tools.

In Safari, open the Preferences window, and on the Advanced pane check the **Show Develop menu in the menu bar** option. After it has been enabled, you can bring up the developer tools by pressing ⌘-OPTION-I.

After enabling or installing the developer tools in your browser, you can right-click any part of the web page and select **Inspect Element** from the context menu to bring up the HTML responsible for that part of the page. This will be helpful when you begin to parse HTML for your web scraping programs.

DON'T USE REGULAR EXPRESSIONS TO PARSE HTML

Locating a specific piece of HTML in a string seems like a perfect case for regular expressions. However, I advise you against it. There are many different ways that HTML can be formatted and still be considered valid HTML, but trying to capture all these possible variations in a regular expression can be tedious and error prone. A module developed specifically for parsing HTML, such as Beautiful Soup, will be less likely to result in bugs.

You can find an extended argument for why you shouldn't to parse HTML with regular expressions at <http://stackoverflow.com/a/1732454/1893164/>.

Using the Developer Tools to Find HTML Elements

Once your program has downloaded a web page using the requests module, you will have the page's HTML content as a single string value. Now you need to figure out which part of the HTML corresponds to the information on the web page you're interested in.

This is where the browser's developer tools can help. Say you want to write a program to pull weather forecast data from <http://weather.gov/>. Before writing any code, do a little research. If you visit the site and search for the 94105 ZIP code, the site will take you to a page showing the forecast for that area.

What if you're interested in scraping the temperature information for that ZIP code? Right-click where it is on the page (or CONTROL-click on OS X) and select **Inspect Element** from the context menu that appears. This will bring up the Developer Tools window, which shows you the HTML that produces this particular part of the web page. Figure 11-5 shows the developer tools open to the HTML of the temperature.

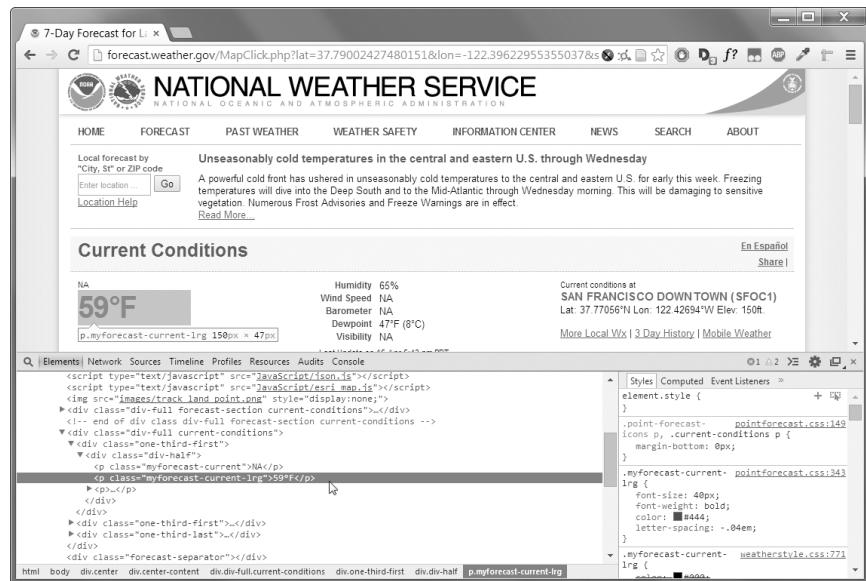


Figure 11-5: Inspecting the element that holds the temperature text with the developer tools

From the developer tools, you can see that the HTML responsible for the temperature part of the web page is `<p class="myforecast-current-lrg">59°F</p>`. This is exactly what you were looking for! It seems that the temperature information is contained inside a `<p>` element with the `myforecast-current-lrg` class. Now that you know what you're looking for, the BeautifulSoup module will help you find it in the string.

Parsing HTML with the BeautifulSoup Module

Beautiful Soup is a module for extracting information from an HTML page (and is much better for this purpose than regular expressions). The BeautifulSoup module's name is `bs4` (for Beautiful Soup, version 4). To install it, you will need to run `pip install beautifulsoup4` from the command line. (Check out Appendix A for instructions on installing third-party modules.) While `beautifulsoup4` is the name used for installation, to import BeautifulSoup you run `import bs4`.

For this chapter, the BeautifulSoup examples will *parse* (that is, analyze and identify the parts of) an HTML file on the hard drive. Open a new file editor window in IDLE, enter the following, and save it as `example.html`. Alternatively, download it from <http://nostarch.com/automatestuff/>.

```
<!-- This is the example.html example file. -->

<html><head><title>The Website Title</title></head>
<body>
<p>Download my <strong>Python</strong> book from <a href="http://
inventwithpython.com">my website</a>.</p>
<p class="slogan">Learn Python the easy way!</p>
<p>By <span id="author">Al Sweigart</span></p>
</body></html>
```

As you can see, even a simple HTML file involves many different tags and attributes, and matters quickly get confusing with complex websites. Thankfully, BeautifulSoup makes working with HTML much easier.

Creating a BeautifulSoup Object from HTML

The `bs4.BeautifulSoup()` function needs to be called with a string containing the HTML it will parse. The `bs4.BeautifulSoup()` function returns a BeautifulSoup object. Enter the following into the interactive shell while your computer is connected to the Internet:

```
>>> import requests, bs4
>>> res = requests.get('http://nostarch.com')
>>> res.raise_for_status()
>>> noStarchSoup = bs4.BeautifulSoup(res.text)
>>> type(noStarchSoup)
<class 'bs4.BeautifulSoup'>
```

This code uses `requests.get()` to download the main page from the No Starch Press website and then passes the `text` attribute of the response to `bs4.BeautifulSoup()`. The BeautifulSoup object that it returns is stored in a variable named `noStarchSoup`.

You can also load an HTML file from your hard drive by passing a `File` object to `bs4.BeautifulSoup()`. Enter the following into the interactive shell (make sure the `example.html` file is in the working directory):

```
>>> exampleFile = open('example.html')
>>> exampleSoup = bs4.BeautifulSoup(exampleFile)
>>> type(exampleSoup)
<class 'bs4.BeautifulSoup'>
```

Once you have a `BeautifulSoup` object, you can use its methods to locate specific parts of an HTML document.

Finding an Element with the `select()` Method

You can retrieve a web page element from a `BeautifulSoup` object by calling the `select()` method and passing a string of a CSS *selector* for the element you are looking for. Selectors are like regular expressions: They specify a pattern to look for, in this case, in HTML pages instead of general text strings.

A full discussion of CSS selector syntax is beyond the scope of this book (there's a good selector tutorial in the resources at <http://nostarch.com/automatestuff/>), but here's a short introduction to selectors. Table 11-2 shows examples of the most common CSS selector patterns.

Table 11-2: Examples of CSS Selectors

Selector passed to the <code>select()</code> method	Will match ...
<code>soup.select('div')</code>	All elements named <code><div></code>
<code>soup.select('#author')</code>	The element with an <code>id</code> attribute of <code>author</code>
<code>soup.select('.notice')</code>	All elements that use a CSS class attribute named <code>notice</code>
<code>soup.select('div span')</code>	All elements named <code></code> that are within an element named <code><div></code>
<code>soup.select('div > span')</code>	All elements named <code></code> that are <i>directly</i> within an element named <code><div></code> , with no other element in between
<code>soup.select('input[name]')</code>	All elements named <code><input></code> that have a <code>name</code> attribute with any value
<code>soup.select('input[type="button"]')</code>	All elements named <code><input></code> that have an attribute named <code>type</code> with value <code>button</code>

The various selector patterns can be combined to make sophisticated matches. For example, `soup.select('p #author')` will match any element that has an `id` attribute of `author`, as long as it is also inside a `<p>` element.

The `select()` method will return a list of `Tag` objects, which is how BeautifulSoup represents an HTML element. The list will contain one `Tag` object for every match in the `BeautifulSoup` object's HTML. `Tag` values can be passed to the `str()` function to show the HTML tags they represent.

Tag values also have an `attrs` attribute that shows all the HTML attributes of the tag as a dictionary. Using the `example.html` file from earlier, enter the following into the interactive shell:

```
>>> import bs4
>>> exampleFile = open('example.html')
>>> exampleSoup = bs4.BeautifulSoup(exampleFile.read())
>>> elems = exampleSoup.select('#author')
>>> type(elems)
<class 'list'>
>>> len(elems)
1
>>> type(elems[0])
<class 'bs4.element.Tag'>
>>> elems[0].getText()
'Al Sweigart'
>>> str(elems[0])
'<span id="author">Al Sweigart</span>'
>>> elems[0].attrs
{'id': 'author'}
```

This code will pull the element with `id="author"` out of our example HTML. We use `select('#author')` to return a list of all the elements with `id="author"`. We store this list of `Tag` objects in the variable `elems`, and `len(elems)` tells us there is one `Tag` object in the list; there was one match. Calling `getText()` on the element returns the element's text, or inner HTML. The text of an element is the content between the opening and closing tags: in this case, 'Al Sweigart'.

Passing the element to `str()` returns a string with the starting and closing tags and the element's text. Finally, `attrs` gives us a dictionary with the element's attribute, '`id`', and the value of the `id` attribute, '`author`'.

You can also pull all the `<p>` elements from the `BeautifulSoup` object. Enter this into the interactive shell:

```
>>> pElems = exampleSoup.select('p')
>>> str(pElems[0])
'<p>Download my <strong>Python</strong> book from <a href="http://
inventwithpython.com">my website</a>.</p>'
>>> pElems[0].getText()
'Download my Python book from my website.'
>>> str(pElems[1])
'<p class="slogan">Learn Python the easy way!</p>'
>>> pElems[1].getText()
'Learn Python the easy way!'
>>> str(pElems[2])
'<p>By <span id="author">Al Sweigart</span></p>'
>>> pElems[2].getText()
'By Al Sweigart'
```

This time, `select()` gives us a list of three matches, which we store in `pElems`. Using `str()` on `pElems[0]`, `pElems[1]`, and `pElems[2]` shows you each element as a string, and using `getText()` on each element shows you its text.

Getting Data from an Element's Attributes

The `get()` method for `Tag` objects makes it simple to access attribute values from an element. The method is passed a string of an attribute name and returns that attribute's value. Using `example.html`, enter the following into the interactive shell:

```
>>> import bs4
>>> soup = bs4.BeautifulSoup(open('example.html'))
>>> spanElem = soup.select('span')[0]
>>> str(spanElem)
'<span id="author">Al Sweigart</span>'
>>> spanElem.get('id')
'author'
>>> spanElem.get('some_nonexistent_addr') == None
True
>>> spanElem.attrs
{'id': 'author'}
```

Here we use `select()` to find any `` elements and then store the first matched element in `spanElem`. Passing the attribute name `'id'` to `get()` returns the attribute's value, `'author'`.

Project: “I’m Feeling Lucky” Google Search

Whenever I search a topic on Google, I don’t look at just one search result at a time. By middle-clicking a search result link (or clicking while holding `CTRL`), I open the first several links in a bunch of new tabs to read later. I search Google often enough that this workflow—opening my browser, searching for a topic, and middle-clicking several links one by one—is tedious. It would be nice if I could simply type a search term on the command line and have my computer automatically open a browser with all the top search results in new tabs. Let’s write a script to do this.

This is what your program does:

- Gets search keywords from the command line arguments.
- Retrieves the search results page.
- Opens a browser tab for each result.

This means your code will need to do the following:

- Read the command line arguments from `sys.argv`.
- Fetch the search result page with the `requests` module.
- Find the links to each search result.
- Call the `webbrowser.open()` function to open the web browser.

Open a new file editor window and save it as `lucky.py`.

Step 1: Get the Command Line Arguments and Request the Search Page

Before coding anything, you first need to know the URL of the search result page. By looking at the browser's address bar after doing a Google search, you can see that the result page has a URL like `https://www.google.com/search?q=SEARCH_TERM_HERE`. The `requests` module can download this page and then you can use Beautiful Soup to find the search result links in the HTML. Finally, you'll use the `webbrowser` module to open those links in browser tabs.

Make your code look like the following:

```
#! python3
# lucky.py - Opens several Google search results.

import requests, sys, webbrowser, bs4

print('Googling...')    # display text while downloading the Google page
res = requests.get('http://google.com/search?q=' + ' '.join(sys.argv[1:]))
res.raise_for_status()

# TODO: Retrieve top search result links.

# TODO: Open a browser tab for each result.
```

The user will specify the search terms using command line arguments when they launch the program. These arguments will be stored as strings in a list in `sys.argv`.

Step 2: Find All the Results

Now you need to use Beautiful Soup to extract the top search result links from your downloaded HTML. But how do you figure out the right selector for the job? For example, you can't just search for all `<a>` tags, because there are lots of links you don't care about in the HTML. Instead, you must inspect the search result page with the browser's developer tools to try to find a selector that will pick out only the links you want.

After doing a Google search for *Beautiful Soup*, you can open the browser's developer tools and inspect some of the link elements on the page. They look incredibly complicated, something like this: `BeautifulSoup: We called him Tortoise because he taught us.`

It doesn't matter that the element looks incredibly complicated. You just need to find the pattern that all the search result links have. But this `<a>` element doesn't have anything that easily distinguishes it from the nonsearch result `<a>` elements on the page.

Make your code look like the following:

```
#! python3
# lucky.py - Opens several google search results.

import requests, sys, webbrowser, bs4

--snip--

# Retrieve top search result links.
soup = bs4.BeautifulSoup(res.text)

# Open a browser tab for each result.
linkElems = soup.select('.r a')
```

If you look up a little from the `<a>` element, though, there is an element like this: `<h3 class="r">`. Looking through the rest of the HTML source, it looks like the `r` class is used only for search result links. You don't have to know what the CSS class `r` is or what it does. You're just going to use it as a marker for the `<a>` element you are looking for. You can create a `BeautifulSoup` object from the downloaded page's HTML text and then use the selector `'.r a'` to find all `<a>` elements that are within an element that has the `r` CSS class.

Step 3: Open Web Browsers for Each Result

Finally, we'll tell the program to open web browser tabs for our results. Add the following to the end of your program:

```
#! python3
# lucky.py - Opens several google search results.

import requests, sys, webbrowser, bs4

--snip--

# Open a browser tab for each result.
linkElems = soup.select('.r a')
numOpen = min(5, len(linkElems))
for i in range(numOpen):
    webbrowser.open('http://google.com' + linkElems[i].get('href'))
```

By default, you open the first five search results in new tabs using the `webbrowser` module. However, the user may have searched for something that turned up fewer than five results. The `soup.select()` call returns a list of all the elements that matched your `'.r a'` selector, so the number of tabs you want to open is either 5 or the length of this list (whichever is smaller).

The built-in Python function `min()` returns the smallest of the integer or float arguments it is passed. (There is also a built-in `max()` function that

returns the largest argument it is passed.) You can use `min()` to find out whether there are fewer than five links in the list and store the number of links to open in a variable named `numOpen`. Then you can run through a `for` loop by calling `range(numOpen)`.

On each iteration of the loop, you use `webbrowser.open()` to open a new tab in the web browser. Note that the `href` attribute's value in the returned `<a>` elements do not have the initial `http://google.com` part, so you have to concatenate that to the `href` attribute's string value.

Now you can instantly open the first five Google results for, say, *Python programming tutorials* by running `lucky python programming tutorials` on the command line! (See Appendix B for how to easily run programs on your operating system.)

Ideas for Similar Programs

The benefit of tabbed browsing is that you can easily open links in new tabs to peruse later. A program that automatically opens several links at once can be a nice shortcut to do the following:

- Open all the product pages after searching a shopping site such as Amazon
- Open all the links to reviews for a single product
- Open the result links to photos after performing a search on a photo site such as Flickr or Imgur

Project: Downloading All XKCD Comics

Blogs and other regularly updating websites usually have a front page with the most recent post as well as a Previous button on the page that takes you to the previous post. Then that post will also have a Previous button, and so on, creating a trail from the most recent page to the first post on the site. If you wanted a copy of the site's content to read when you're not online, you could manually navigate over every page and save each one. But this is pretty boring work, so let's write a program to do it instead.

XKCD is a popular geek webcomic with a website that fits this structure (see Figure 11-6). The front page at `http://xkcd.com/` has a Prev button that guides the user back through prior comics. Downloading each comic by hand would take forever, but you can write a script to do this in a couple of minutes.

Here's what your program does:

- Loads the XKCD home page.
- Saves the comic image on that page.
- Follows the Previous Comic link.
- Repeats until it reaches the first comic.

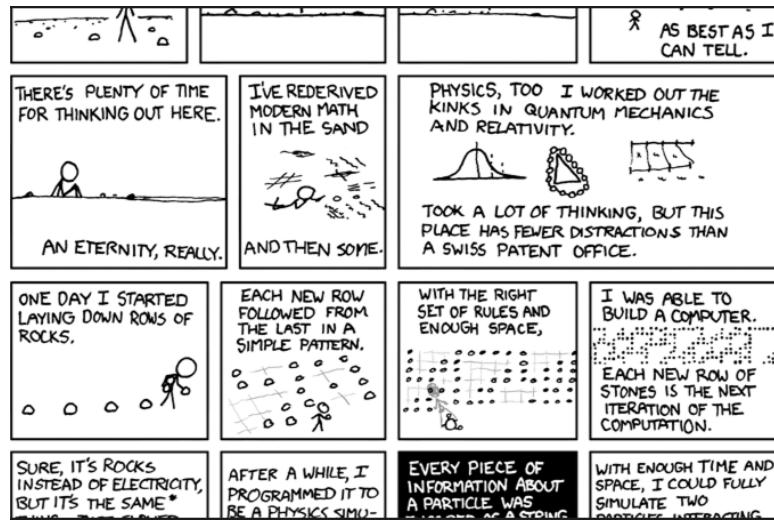


Figure 11-6: XKCD, “a webcomic of romance, sarcasm, math, and language”

This means your code will need to do the following:

- Download pages with the `requests` module.
- Find the URL of the comic image for a page using `Beautiful Soup`.
- Download and save the comic image to the hard drive with `iter_content()`.
- Find the URL of the Previous Comic link, and repeat.

Open a new file editor window and save it as `downloadXkcd.py`.

Step 1: Design the Program

If you open the browser’s developer tools and inspect the elements on the page, you’ll find the following:

- The URL of the comic’s image file is given by the `href` attribute of an `` element.
- The `` element is inside a `<div id="comic">` element.
- The Prev button has a `rel` HTML attribute with the value `prev`.
- The first comic’s Prev button links to the `http://xkcd.com/#` URL, indicating that there are no more previous pages.

Make your code look like the following:

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

url = 'http://xkcd.com/' # starting url
os.makedirs('xkcd', exist_ok=True) # store comics in ./xkcd
```

```
while not url.endswith('#'):
    # TODO: Download the page.

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')


---


```

You'll have a `url` variable that starts with the value '`http://xkcd.com`' and repeatedly update it (in a `for` loop) with the URL of the current page's Prev link. At every step in the loop, you'll download the comic at `url`. You'll know to end the loop when `url` ends with '#'.

You will download the image files to a folder in the current working directory named `xkcd`. The call `os.makedirs()` ensures that this folder exists, and the `exist_ok=True` keyword argument prevents the function from throwing an exception if this folder already exists. The rest of the code is just comments that outline the rest of your program.

Step 2: Download the Web Page

Let's implement the code for downloading the page. Make your code look like the following:

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

url = 'http://xkcd.com'                      # starting url
os.makedirs('xkcd', exist_ok=True)      # store comics in ./xkcd
while not url.endswith('#'):
    # Download the page.
    print('Downloading page %s...' % url)
    res = requests.get(url)
    res.raise_for_status()

    soup = bs4.BeautifulSoup(res.text)

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')


---


```

First, print url so that the user knows which URL the program is about to download; then use the requests module’s request.get() function to download it. As always, you immediately call the Response object’s raise_for_status() method to throw an exception and end the program if something went wrong with the download. Otherwise, you create a BeautifulSoup object from the text of the downloaded page.

Step 3: Find and Download the Comic Image

Make your code look like the following:

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

--snip--

# Find the URL of the comic image.
comicElem = soup.select('#comic img')
if comicElem == []:
    print('Could not find comic image.')
else:
    comicUrl = comicElem[0].get('src')
    # Download the image.
    print('Downloading image %s...' % (comicUrl))
    res = requests.get(comicUrl)
    res.raise_for_status()

# TODO: Save the image to ./xkcd.

# TODO: Get the Prev button's url.

print('Done.')
```

From inspecting the XKCD home page with your developer tools, you know that the `` element for the comic image is inside a `<div>` element with the `id` attribute set to `comic`, so the selector `'#comic img'` will get you the correct `` element from the BeautifulSoup object.

A few XKCD pages have special content that isn’t a simple image file. That’s fine; you’ll just skip those. If your selector doesn’t find any elements, then `soup.select('#comic img')` will return a blank list. When that happens, the program can just print an error message and move on without downloading the image.

Otherwise, the selector will return a list containing one `` element. You can get the `src` attribute from this `` element and pass it to `requests.get()` to download the comic’s image file.

Step 4: Save the Image and Find the Previous Comic

Make your code look like the following:

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

--snip--

    # Save the image to ./xkcd.
    imageFile = open(os.path.join('xkcd', os.path.basename(comicUrl)), 'wb')
    for chunk in res.iter_content(100000):
        imageFile.write(chunk)
    imageFile.close()

    # Get the Prev button's url.
    prevLink = soup.select('a[rel="prev"]')[0]
    url = 'http://xkcd.com' + prevLink.get('href')

print('Done.')
```

At this point, the image file of the comic is stored in the `res` variable. You need to write this image data to a file on the hard drive.

You'll need a filename for the local image file to pass to `open()`. The `comicUrl` will have a value like '`http://imgs.xkcd.com/comics/heartbleed_explanation.png`'—which you might have noticed looks a lot like a file path. And in fact, you can call `os.path.basename()` with `comicUrl`, and it will return just the last part of the URL, '`heartbleed_explanation.png`'. You can use this as the filename when saving the image to your hard drive. You join this name with the name of your `xkcd` folder using `os.path.join()` so that your program uses backslashes (`\`) on Windows and forward slashes (`/`) on OS X and Linux. Now that you finally have the filename, you can call `open()` to open a new file in '`wb`' “write binary” mode.

Remember from earlier in this chapter that to save files you've downloaded using Requests, you need to loop over the return value of the `iter_content()` method. The code in the `for` loop writes out chunks of the image data (at most 100,000 bytes each) to the file and then you close the file. The image is now saved to your hard drive.

Afterward, the selector '`a[rel="prev"]`' identifies the `<a>` element with the `rel` attribute set to `prev`, and you can use this `<a>` element's `href` attribute to get the previous comic's URL, which gets stored in `url`. Then the `while` loop begins the entire download process again for this comic.

The output of this program will look like this:

```
Downloading page http://xkcd.com...
Downloading image http://imgs.xkcd.com/comics/phone_alarm.png...
Downloading page http://xkcd.com/1358/...
```

```
Downloading image http://imgs.xkcd.com/comics/nro.png...
Downloading page http://xkcd.com/1357/...
Downloading image http://imgs.xkcd.com/comics/free_speech.png...
Downloading page http://xkcd.com/1356/...
Downloading image http://imgs.xkcd.com/comics/orbital_mechanics.png...
Downloading page http://xkcd.com/1355/...
Downloading image http://imgs.xkcd.com/comics/airplane_message.png...
Downloading page http://xkcd.com/1354/...
Downloading image http://imgs.xkcd.com/comics/heartbleed_explanation.png...
--snip--
```

This project is a good example of a program that can automatically follow links in order to scrape large amounts of data from the Web. You can learn about BeautifulSoup's other features from its documentation at <http://www.crummy.com/software/BeautifulSoup/bs4/doc/>.

Ideas for Similar Programs

Downloading pages and following links are the basis of many web crawling programs. Similar programs could also do the following:

- Back up an entire site by following all of its links.
- Copy all the messages off a web forum.
- Duplicate the catalog of items for sale on an online store.

The requests and BeautifulSoup modules are great as long as you can figure out the URL you need to pass to `requests.get()`. However, sometimes this isn't so easy to find. Or perhaps the website you want your program to navigate requires you to log in first. The selenium module will give your programs the power to perform such sophisticated tasks.

Controlling the Browser with the selenium Module

The selenium module lets Python directly control the browser by programmatically clicking links and filling in login information, almost as though there is a human user interacting with the page. Selenium allows you to interact with web pages in a much more advanced way than Requests and BeautifulSoup; but because it launches a web browser, it is a bit slower and hard to run in the background if, say, you just need to download some files from the Web.

Appendix A has more detailed steps on installing third-party modules.

Starting a Selenium-Controlled Browser

For these examples, you'll need the Firefox web browser. This will be the browser that you control. If you don't already have Firefox, you can download it for free from <http://getfirefox.com/>.

Importing the modules for Selenium is slightly tricky. Instead of `import selenium`, you need to run `from selenium import webdriver`. (The exact reason why the `selenium` module is set up this way is beyond the scope of this book.) After that, you can launch the Firefox browser with Selenium. Enter the following into the interactive shell:

```
>>> from selenium import webdriver  
>>> browser = webdriver.Firefox()  
>>> type(browser)  
<class 'selenium.webdriver.firefox.webdriver.WebDriver'>  
>>> browser.get('http://inventwithpython.com')
```

You'll notice when `webdriver.Firefox()` is called, the Firefox web browser starts up. Calling `type()` on the value `webdriver.Firefox()` reveals it's of the `WebDriver` data type. And calling `browser.get('http://inventwithpython.com')` directs the browser to `http://inventwithpython.com/`. Your browser should look something like Figure 11-7.



Figure 11-7: After calling `webdriver.Firefox()` and `get()` in IDLE, the Firefox browser appears.

Finding Elements on the Page

WebDriver objects have quite a few methods for finding elements on a page. They are divided into the `find_element_*` and `find_elements_*` methods. The `find_element_*` methods return a single WebElement object, representing the first element on the page that matches your query. The `find_elements_*` methods return a list of WebElement_* objects for *every* matching element on the page.

Table 11-3 shows several examples of `find_element_*` and `find_elements_*` methods being called on a WebDriver object that's stored in the variable `browser`.

Table 11-3: Selenium's WebDriver Methods for Finding Elements

Method name	WebElement object/list returned
<code>browser.find_element_by_class_name(name)</code> <code>browser.find_elements_by_class_name(name)</code>	Elements that use the CSS class <code>name</code>
<code>browser.find_element_by_css_selector(selector)</code> <code>browser.find_elements_by_css_selector(selector)</code>	Elements that match the CSS <code>selector</code>
<code>browser.find_element_by_id(id)</code> <code>browser.find_elements_by_id(id)</code>	Elements with a matching <code>id</code> attribute value
<code>browser.find_element_by_link_text(text)</code> <code>browser.find_elements_by_link_text(text)</code>	<code><a></code> elements that completely match the <code>text</code> provided
<code>browser.find_element_by_partial_link_text(text)</code> <code>browser.find_elements_by_partial_link_text(text)</code>	<code><a></code> elements that contain the <code>text</code> provided
<code>browser.find_element_by_name(name)</code> <code>browser.find_elements_by_name(name)</code>	Elements with a matching <code>name</code> attribute value
<code>browser.find_element_by_tag_name(name)</code> <code>browser.find_elements_by_tag_name(name)</code>	Elements with a matching tag <code>name</code> (case insensitive; an <code><a></code> element is matched by 'a' and 'A')

Except for the `*_by_tag_name()` methods, the arguments to all the methods are case sensitive. If no elements exist on the page that match what the method is looking for, the `selenium` module raises a `NoSuchElement` exception. If you do not want this exception to crash your program, add `try` and `except` statements to your code.

Once you have the `WebElement` object, you can find out more about it by reading the attributes or calling the methods in Table 11-4.

Table 11-4: WebElement Attributes and Methods

Attribute or method	Description
<code>tag_name</code>	The tag name, such as 'a' for an <code><a></code> element
<code>get_attribute(name)</code>	The value for the element's <code>name</code> attribute
<code>text</code>	The text within the element, such as 'hello' in <code>hello</code>
<code>clear()</code>	For text field or text area elements, clears the text typed into it
<code>is_displayed()</code>	Returns True if the element is visible; otherwise returns False
<code>is_enabled()</code>	For input elements, returns True if the element is enabled; otherwise returns False
<code>is_selected()</code>	For checkbox or radio button elements, returns True if the element is selected; otherwise returns False
<code>location</code>	A dictionary with keys 'x' and 'y' for the position of the element in the page

For example, open a new file editor and enter the following program:

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://inventwithpython.com')
```

```
try:  
    elem = browser.find_element_by_class_name('bookcover')  
    print('Found <%s> element with that class name!' % (elem.tag_name))  
except:  
    print('Was not able to find an element with that name.')

---


```

Here we open Firefox and direct it to a URL. On this page, we try to find elements with the class name 'bookcover', and if such an element is found, we print its tag name using the `tag_name` attribute. If no such element was found, we print a different message.

This program will output the following:

```
Found <img> element with that class name!
```

We found an element with the class name 'bookcover' and the tag name 'img'.

Clicking the Page

`WebElement` objects returned from the `find_element_*` and `find_elements_*` methods have a `click()` method that simulates a mouse click on that element. This method can be used to follow a link, make a selection on a radio button, click a Submit button, or trigger whatever else might happen when the element is clicked by the mouse. For example, enter the following into the interactive shell:

```
>>> from selenium import webdriver  
>>> browser = webdriver.Firefox()  
>>> browser.get('http://inventwithpython.com')  
>>> linkElem = browser.find_element_by_link_text('Read It Online')  
>>> type(linkElem)  
<class 'selenium.webdriver.remote.webelement.WebElement'>  
>>> linkElem.click()      # follows the "Read It Online" link
```

This opens Firefox to `http://inventwithpython.com/`, gets the `WebElement` object for the `<a>` element with the text *Read It Online*, and then simulates clicking that `<a>` element. It's just like if you clicked the link yourself; the browser then follows that link.

Filling Out and Submitting Forms

Sending keystrokes to text fields on a web page is a matter of finding the `<input>` or `<textarea>` element for that text field and then calling the `send_keys()` method. For example, enter the following into the interactive shell:

```
>>> from selenium import webdriver  
>>> browser = webdriver.Firefox()  
>>> browser.get('http://gmail.com')  
>>> emailElem = browser.find_element_by_id('Email')  
>>> emailElem.send_keys('not_my_real_email@gmail.com')  
>>> passwordElem = browser.find_element_by_id('Passwd')
```

```
>>> passwordElem.send_keys('12345')
>>> passwordElem.submit()
```

As long as Gmail hasn't changed the `id` of the Username and Password text fields since this book was published, the previous code will fill in those text fields with the provided text. (You can always use the browser's inspector to verify the `id`.) Calling the `submit()` method on any element will have the same result as clicking the Submit button for the form that element is in. (You could have just as easily called `emailElem.submit()`, and the code would have done the same thing.)

Sending Special Keys

Selenium has a module for keyboard keys that are impossible to type into a string value, which function much like escape characters. These values are stored in attributes in the `selenium.webdriver.common.keys` module. Since that is such a long module name, it's much easier to run `from selenium.webdriver.common.keys import Keys` at the top of your program; if you do, then you can simply write `Keys` anywhere you'd normally have to write `selenium.webdriver.common.keys`. Table 11-5 lists the commonly used `Keys` variables.

Table 11-5: Commonly Used Variables in the `selenium.webdriver.common.keys` Module

Attributes	Meanings
<code>Keys.DOWN</code> , <code>Keys.UP</code> , <code>Keys.LEFT</code> , <code>Keys.RIGHT</code>	The keyboard arrow keys
<code>Keys.ENTER</code> , <code>Keys.RETURN</code>	The ENTER and RETURN keys
<code>Keys.HOME</code> , <code>Keys.END</code> , <code>Keys.PAGE_DOWN</code> , <code>Keys.PAGE_UP</code>	The HOME, END, PAGEDOWN, and PAGEUP keys
<code>Keys.ESCAPE</code> , <code>Keys.BACK_SPACE</code> , <code>Keys.DELETE</code>	The ESC, BACKSPACE, and DELETE keys
<code>Keys.F1</code> , <code>Keys.F2</code> , . . . , <code>Keys.F12</code>	The F1 to F12 keys at the top of the keyboard
<code>Keys.TAB</code>	The TAB key

For example, if the cursor is not currently in a text field, pressing the HOME and END keys will scroll the browser to the top and bottom of the page, respectively. Enter the following into the interactive shell, and notice how the `send_keys()` calls scroll the page:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.common.keys import Keys
>>> browser = webdriver.Firefox()
>>> browser.get('http://nostarch.com')
>>> htmlElem = browser.find_element_by_tag_name('html')
>>> htmlElem.send_keys(Keys.END)      # scrolls to bottom
>>> htmlElem.send_keys(Keys.HOME)    # scrolls to top
```

The `<html>` tag is the base tag in HTML files: The full content of the HTML file is enclosed within the `<html>` and `</html>` tags. Calling `browser.find_element_by_tag_name('html')` is a good place to send keys to the general web page. This would be useful if, for example, new content is loaded once you've scrolled to the bottom of the page.

Clicking Browser Buttons

Selenium can simulate clicks on various browser buttons as well through the following methods:

- `browser.back()` Clicks the Back button.
- `browser.forward()` Clicks the Forward button.
- `browser.refresh()` Clicks the Refresh/Reload button.
- `browser.quit()` Clicks the Close Window button.

More Information on Selenium

Selenium can do much more beyond the functions described here. It can modify your browser's cookies, take screenshots of web pages, and run custom JavaScript. To learn more about these features, you can visit the Selenium documentation at <http://selenium-python.readthedocs.org/>.

Summary

Most boring tasks aren't limited to the files on your computer. Being able to programmatically download web pages will extend your programs to the Internet. The `requests` module makes downloading straightforward, and with some basic knowledge of HTML concepts and selectors, you can utilize the `BeautifulSoup` module to parse the pages you download.

But to fully automate any web-based tasks, you need direct control of your web browser through the `selenium` module. The `selenium` module will allow you to log in to websites and fill out forms automatically. Since a web browser is the most common way to send and receive information over the Internet, this is a great ability to have in your programmer toolkit.

Practice Questions

1. Briefly describe the differences between the `webbrowser`, `requests`, `BeautifulSoup`, and `selenium` modules.
2. What type of object is returned by `requests.get()`? How can you access the downloaded content as a string value?
3. What `Requests` method checks that the download worked?
4. How can you get the HTTP status code of a `Requests` response?
5. How do you save a `Requests` response to a file?

6. What is the keyboard shortcut for opening a browser's developer tools?
7. How can you view (in the developer tools) the HTML of a specific element on a web page?
8. What is the CSS selector string that would find the element with an `id` attribute of `main`?
9. What is the CSS selector string that would find the elements with a CSS class of `highlight`?
10. What is the CSS selector string that would find all the `<div>` elements inside another `<div>` element?
11. What is the CSS selector string that would find the `<button>` element with a `value` attribute set to `favorite`?
12. Say you have a Beautiful Soup `Tag` object stored in the variable `spam` for the element `<div>Hello world!</div>`. How could you get a string '`Hello world!`' from the `Tag` object?
13. How would you store all the attributes of a Beautiful Soup `Tag` object in a variable named `linkElem`?
14. Running `import selenium` doesn't work. How do you properly import the `selenium` module?
15. What's the difference between the `find_element_*` and `find_elements_*` methods?
16. What methods do Selenium's `WebElement` objects have for simulating mouse clicks and keyboard keys?
17. You could call `send_keys(Keys.ENTER)` on the Submit button's `WebElement` object, but what is an easier way to submit a form with Selenium?
18. How can you simulate clicking a browser's Forward, Back, and Refresh buttons with Selenium?

Practice Projects

For practice, write programs to do the following tasks.

Command Line Emailer

Write a program that takes an email address and string of text on the command line and then, using Selenium, logs into your email account and sends an email of the string to the provided address. (You might want to set up a separate email account for this program.)

This would be a nice way to add a notification feature to your programs. You could also write a similar program to send messages from a Facebook or Twitter account.

Image Site Downloader

Write a program that goes to a photo-sharing site like Flickr or Imgur, searches for a category of photos, and then downloads all the resulting images. You could write a program that works with any photo site that has a search feature.

2048

2048 is a simple game where you combine tiles by sliding them up, down, left, or right with the arrow keys. You can actually get a fairly high score by repeatedly sliding in an up, right, down, and left pattern over and over again. Write a program that will open the game at <https://gabrielicirulli.github.io/2048/> and keep sending up, right, down, and left keystrokes to automatically play the game.

Link Verification

Write a program that, given the URL of a web page, will attempt to download every linked page on the page. The program should flag any pages that have a 404 “Not Found” status code and print them out as broken links.

12

WORKING WITH EXCEL SPREADSHEETS



Excel is a popular and powerful spreadsheet application for Windows. The `openpyxl` module allows your Python programs to read and modify Excel spreadsheet files. For example, you might have the boring task of copying certain data from one spreadsheet and pasting it into another one. Or you might have to go through thousands of rows and pick out just a handful of them to make small edits based on some criteria. Or you might have to look through hundreds of spreadsheets of department budgets, searching for any that are in the red. These are exactly the sort of boring, mindless spreadsheet tasks that Python can do for you.

Although Excel is proprietary software from Microsoft, there are free alternatives that run on Windows, OS X, and Linux. Both LibreOffice Calc and OpenOffice Calc work with Excel's `.xlsx` file format for spreadsheets, which means the `openpyxl` module can work on spreadsheets from these applications as well. You can download the software from <https://www.libreoffice.org/> and <http://www.openoffice.org/>, respectively. Even if you already have

Excel installed on your computer, you may find these programs easier to use. The screenshots in this chapter, however, are all from Excel 2010 on Windows 7.

Excel Documents

First, let's go over some basic definitions: An Excel spreadsheet document is called a *workbook*. A single workbook is saved in a file with the *.xlsx* extension. Each workbook can contain multiple *sheets* (also called *worksheets*). The sheet the user is currently viewing (or last viewed before closing Excel) is called the *active sheet*.

Each sheet has *columns* (addressed by letters starting at *A*) and *rows* (addressed by numbers starting at *1*). A box at a particular column and row is called a *cell*. Each cell can contain a number or text value. The grid of cells with data makes up a sheet.

Installing the openpyxl Module

Python does not come with OpenPyXL, so you'll have to install it. Follow the instructions for installing third-party modules in Appendix A; the name of the module is *openpyxl*. To test whether it is installed correctly, enter the following into the interactive shell:

```
>>> import openpyxl
```

If the module was correctly installed, this should produce no error messages. Remember to import the *openpyxl* module before running the interactive shell examples in this chapter, or you'll get a `NameError: name 'openpyxl' is not defined` error.

This book covers version 2.1.4 of OpenPyXL, but new versions are regularly released by the OpenPyXL team. Don't worry, though: New versions should stay backward compatible with the instructions in this book for quite some time. If you have a newer version and want to see what additional features may be available to you, you can check out the full documentation for OpenPyXL at <http://openpyxl.readthedocs.org/>.

Reading Excel Documents

The examples in this chapter will use a spreadsheet named *example.xlsx* stored in the root folder. You can either create the spreadsheet yourself or download it from <http://nostarch.com/automatestuff/>. Figure 12-1 shows the tabs for the three default sheets named *Sheet1*, *Sheet2*, and *Sheet3* that Excel automatically provides for new workbooks. (The number of default sheets created may vary between operating systems and spreadsheet programs.)



Figure 12-1: The tabs for a workbook's sheets are in the lower-left corner of Excel.

Sheet 1 in the example file should look like Table 12-1. (If you didn't download *example.xlsx* from the website, you should enter this data into the sheet yourself.)

Table 12-1: The *example.xlsx* Spreadsheet

	A	B	C
1	4/5/2015 1:34:02 PM	Apples	73
2	4/5/2015 3:41:23 AM	Cherries	85
3	4/6/2015 12:46:51 PM	Pears	14
4	4/8/2015 8:59:43 AM	Oranges	52
5	4/10/2015 2:07:00 AM	Apples	152
6	4/10/2015 6:10:37 PM	Bananas	23
7	4/10/2015 2:40:46 AM	Strawberries	98

Now that we have our example spreadsheet, let's see how we can manipulate it with the `openpyxl` module.

Opening Excel Documents with OpenPyXL

Once you've imported the `openpyxl` module, you'll be able to use the `openpyxl.load_workbook()` function. Enter the following into the interactive shell:

```
>>> import openpyxl  
>>> wb = openpyxl.load_workbook('example.xlsx')  
>>> type(wb)  
<class 'openpyxl.workbook.workbook.Workbook'>
```

The `openpyxl.load_workbook()` function takes in the filename and returns a value of the `workbook` data type. This `Workbook` object represents the Excel file, a bit like how a `File` object represents an opened text file.

Remember that *example.xlsx* needs to be in the current working directory in order for you to work with it. You can find out what the current working directory is by importing `os` and using `os.getcwd()`, and you can change the current working directory using `os.chdir()`.

Getting Sheets from the Workbook

You can get a list of all the sheet names in the workbook by calling the `get_sheet_names()` method. Enter the following into the interactive shell:

```
>>> import openpyxl  
>>> wb = openpyxl.load_workbook('example.xlsx')  
>>> wb.get_sheet_names()  
['Sheet1', 'Sheet2', 'Sheet3']  
>>> sheet = wb.get_sheet_by_name('Sheet3')  
>>> sheet  
<Worksheet "Sheet3">  
>>> type(sheet)  
<class 'openpyxl.worksheet.worksheet.Worksheet'>  
>>> sheet.title  
'Sheet3'  
>>> anotherSheet = wb.get_active_sheet()  
>>> anotherSheet  
<Worksheet "Sheet1">
```

Each sheet is represented by a `Worksheet` object, which you can obtain by passing the sheet name string to the `get_sheet_by_name()` workbook method. Finally, you can call the `get_active_sheet()` method of a `Workbook` object to get the workbook's active sheet. The active sheet is the sheet that's on top when the workbook is opened in Excel. Once you have the `Worksheet` object, you can get its name from the `title` attribute.

Getting Cells from the Sheets

Once you have a `Worksheet` object, you can access a `Cell` object by its name. Enter the following into the interactive shell:

```
>>> import openpyxl  
>>> wb = openpyxl.load_workbook('example.xlsx')  
>>> sheet = wb.get_sheet_by_name('Sheet1')  
>>> sheet['A1']  
<Cell Sheet1.A1>  
>>> sheet['A1'].value  
datetime.datetime(2015, 4, 5, 13, 34, 2)  
>>> c = sheet['B1']  
>>> c.value  
'Apples'  
>>> 'Row ' + str(c.row) + ', Column ' + c.column + ' is ' + c.value  
'Row 1, Column B is Apples'  
>>> 'Cell ' + c.coordinate + ' is ' + c.value  
'Cell B1 is Apples'  
>>> sheet['C1'].value
```

73

The `Cell` object has a `value` attribute that contains, unsurprisingly, the value stored in that cell. `Cell` objects also have `row`, `column`, and `coordinate` attributes that provide location information for the cell.

Here, accessing the `value` attribute of our `Cell` object for cell B1 gives us the string 'Apples'. The `row` attribute gives us the integer 1, the `column` attribute gives us 'B', and the `coordinate` attribute gives us 'B1'.

OpenPyXL will automatically interpret the dates in column A and return them as `datetime` values rather than strings. The `datetime` data type is explained further in Chapter 16.

Specifying a column by letter can be tricky to program, especially because after column Z, the columns start by using two letters: AA, AB, AC, and so on. As an alternative, you can also get a cell using the sheet's `cell()` method and passing integers for its `row` and `column` keyword arguments. The first row or column integer is 1, not 0. Continue the interactive shell example by entering the following:

```
>>> sheet.cell(row=1, column=2)
<Cell Sheet1.B1>
>>> sheet.cell(row=1, column=2).value
'Apples'
>>> for i in range(1, 8, 2):
    print(i, sheet.cell(row=i, column=2).value)

1 Apples
3 Pears
5 Apples
7 Strawberries
```

As you can see, using the sheet's `cell()` method and passing it `row=1` and `column=2` gets you a `Cell` object for cell B1, just like specifying `sheet['B1']` did. Then, using the `cell()` method and its keyword arguments, you can write a `for` loop to print the values of a series of cells.

Say you want to go down column B and print the value in every cell with an odd row number. By passing 2 for the `range()` function's "step" parameter, you can get cells from every second row (in this case, all the odd-numbered rows). The `for` loop's `i` variable is passed for the `row` keyword argument to the `cell()` method, while 2 is always passed for the `column` keyword argument. Note that the integer 2, not the string 'B', is passed.

You can determine the size of the sheet with the `Worksheet` object's `get_highest_row()` and `get_highest_column()` methods. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> sheet.get_highest_row()
7
>>> sheet.get_highest_column()
3
```

Note that the `get_highest_column()` method returns an integer rather than the letter that appears in Excel.

Converting Between Column Letters and Numbers

To convert from letters to numbers, call the `openpyxl.cell.column_index_from_string()` function. To convert from numbers to letters, call the `openpyxl.cell.get_column_letter()` function. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> from openpyxl.cell import get_column_letter, column_index_from_string
>>> get_column_letter(1)
'A'
>>> get_column_letter(2)
'B'
>>> get_column_letter(27)
'AA'
>>> get_column_letter(900)
'AHP'
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> get_column_letter(sheet.get_highest_column())
'C'
>>> column_index_from_string('A')
1
>>> column_index_from_string('AA')
27
```

After you import these two functions from the `openpyxl.cell` module, you can call `get_column_letter()` and pass it an integer like 27 to figure out what the letter name of the 27th column is. The function `column_index_string()` does the reverse: You pass it the letter name of a column, and it tells you what number that column is. You don't need to have a workbook loaded to use these functions. If you want, you can load a workbook, get a `Worksheet` object, and call a `Worksheet` object method like `get_highest_column()` to get an integer. Then, you can pass that integer to `get_column_letter()`.

Getting Rows and Columns from the Sheets

You can slice `Worksheet` objects to get all the `Cell` objects in a row, column, or rectangular area of the spreadsheet. Then you can loop over all the cells in the slice. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> tuple(sheet['A1':'C3'])
((<Cell Sheet1.A1>, <Cell Sheet1.B1>, <Cell Sheet1.C1>), (<Cell Sheet1.A2>,
<Cell Sheet1.B2>, <Cell Sheet1.C2>), (<Cell Sheet1.A3>, <Cell Sheet1.B3>,
<Cell Sheet1.C3>))
❶ >>> for rowOfCellObjects in sheet['A1':'C3']:
❷     for cellObj in rowOfCellObjects:
         print(cellObj.coordinate, cellObj.value)
     print('--- END OF ROW ---')
```

```
A1 2015-04-05 13:34:02
B1 Apples
C1 73
--- END OF ROW ---
A2 2015-04-05 03:41:23
B2 Cherries
C2 85
--- END OF ROW ---
A3 2015-04-06 12:46:51
B3 Pears
C3 14
--- END OF ROW ---
```

Here, we specify that we want the `Cell` objects in the rectangular area from A1 to C3, and we get a `Generator` object containing the `Cell` objects in that area. To help us visualize this `Generator` object, we can use `tuple()` on it to display its `Cell` objects in a tuple.

This tuple contains three tuples: one for each row, from the top of the desired area to the bottom. Each of these three inner tuples contains the `Cell` objects in one row of our desired area, from the leftmost cell to the right. So overall, our slice of the sheet contains all the `Cell` objects in the area from A1 to C3, starting from the top-left cell and ending with the bottom-right cell.

To print the values of each cell in the area, we use two `for` loops. The outer `for` loop goes over each row in the slice ❶. Then, for each row, the nested `for` loop goes through each cell in that row ❷.

To access the values of cells in a particular row or column, you can also use a `Worksheet` object's `rows` and `columns` attribute. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_active_sheet()
>>> sheet.columns[1]
(<Cell Sheet1.B1>, <Cell Sheet1.B2>, <Cell Sheet1.B3>, <Cell Sheet1.B4>,
<Cell Sheet1.B5>, <Cell Sheet1.B6>, <Cell Sheet1.B7>)
>>> for cellObj in sheet.columns[1]:
    print(cellObj.value)

Apples
Cherries
Pears
Oranges
Apples
Bananas
Strawberries
```

Using the `rows` attribute on a `Worksheet` object will give you a tuple of tuples. Each of these inner tuples represents a row, and contains the `Cell` objects in that row. The `columns` attribute also gives you a tuple of tuples, with each of the inner tuples containing the `Cell` objects in a particular

column. For `example.xlsx`, since there are 7 rows and 3 columns, `rows` gives us a tuple of 7 tuples (each containing 3 `Cell` objects), and `columns` gives us a tuple of 3 tuples (each containing 7 `Cell` objects).

To access one particular tuple, you can refer to it by its index in the larger tuple. For example, to get the tuple that represents column B, you use `sheet.columns[1]`. To get the tuple containing the `Cell` objects in column A, you'd use `sheet.columns[0]`. Once you have a tuple representing one row or column, you can loop through its `Cell` objects and print their values.

Workbooks, Sheets, Cells

As a quick review, here's a rundown of all the functions, methods, and data types involved in reading a cell out of a spreadsheet file:

1. Import the `openpyxl` module.
2. Call the `openpyxl.load_workbook()` function.
3. Get a `Workbook` object.
4. Call the `get_active_sheet()` or `get_sheet_by_name()` workbook method.
5. Get a `Worksheet` object.
6. Use indexing or the `cell()` sheet method with `row` and `column` keyword arguments.
7. Get a `Cell` object.
8. Read the `Cell` object's `value` attribute.

Project: Reading Data from a Spreadsheet

Say you have a spreadsheet of data from the 2010 US Census and you have the boring task of going through its thousands of rows to count both the total population and the number of census tracts for each county. (A census tract is simply a geographic area defined for the purposes of the census.) Each row represents a single census tract. We'll name the spreadsheet file `censuspopdata.xlsx`, and you can download it from <http://nostarch.com/automatestuff/>. Its contents look like Figure 12-2.

	A	B	C	D	E
1	CensusTract	State	County	POP2010	
9841	06075010500	CA	San Francisco	2685	
9842	06075010600	CA	San Francisco	3894	
9843	06075010700	CA	San Francisco	5592	
9844	06075010800	CA	San Francisco	4578	
9845	06075010900	CA	San Francisco	4320	
9846	06075011000	CA	San Francisco	4827	
9847	06075011100	CA	San Francisco	5164	

Figure 12-2: The `censuspopdata.xlsx` spreadsheet

Even though Excel can calculate the sum of multiple selected cells, you'd still have to select the cells for each of the 3,000-plus counties. Even if it takes just a few seconds to calculate a county's population by hand, this would take hours to do for the whole spreadsheet.

In this project, you'll write a script that can read from the census spreadsheet file and calculate statistics for each county in a matter of seconds.

This is what your program does:

- Reads the data from the Excel spreadsheet.
- Counts the number of census tracts in each county.
- Counts the total population of each county.
- Prints the results.

This means your code will need to do the following:

- Open and read the cells of an Excel document with the `openpyxl` module.
- Calculate all the tract and population data and store it in a data structure.
- Write the data structure to a text file with the `.py` extension using the `pprint` module.

Step 1: Read the Spreadsheet Data

There is just one sheet in the `censuspopdata.xlsx` spreadsheet, named '`Population by Census Tract`', and each row holds the data for a single census tract. The columns are the tract number (A), the state abbreviation (B), the county name (C), and the population of the tract (D).

Open a new file editor window and enter the following code. Save the file as `readCensusExcel.py`.

```
#! python3
# readCensusExcel.py - Tabulates population and number of census tracts for
# each county.

❶ import openpyxl, pprint
print('Opening workbook...')
❷ wb = openpyxl.load_workbook('censuspopdata.xlsx')
❸ sheet = wb.get_sheet_by_name('Population by Census Tract')
countyData = {}

# TODO: Fill in countyData with each county's population and tracts.
print('Reading rows...')
❹ for row in range(2, sheet.get_highest_row() + 1):
    # Each row in the spreadsheet has data for one census tract.
    state = sheet['B' + str(row)].value
    county = sheet['C' + str(row)].value
    pop = sheet['D' + str(row)].value

# TODO: Open a new text file and write the contents of countyData to it.
```

This code imports the `openpyxl` module, as well as the `pprint` module that you'll use to print the final county data ❶. Then it opens the `censuspopdata.xlsx` file ❷, gets the sheet with the census data ❸, and begins iterating over its rows ❹.

Note that you've also created a variable named `countyData`, which will contain the populations and number of tracts you calculate for each county. Before you can store anything in it, though, you should determine exactly how you'll structure the data inside it.

Step 2: Populate the Data Structure

The data structure stored in `countyData` will be a dictionary with state abbreviations as its keys. Each state abbreviation will map to another dictionary, whose keys are strings of the county names in that state. Each county name will in turn map to a dictionary with just two keys, 'tracts' and 'pop'. These keys map to the number of census tracts and population for the county. For example, the dictionary will look similar to this:

```
{'AK': {'Aleutians East': {'pop': 3141, 'tracts': 1},  
        'Aleutians West': {'pop': 5561, 'tracts': 2},  
        'Anchorage': {'pop': 291826, 'tracts': 55},  
        'Bethel': {'pop': 17013, 'tracts': 3},  
        'Bristol Bay': {'pop': 997, 'tracts': 1},  
        --snip--
```

If the previous dictionary were stored in `countyData`, the following expressions would evaluate like this:

```
>>> countyData['AK']['Anchorage']['pop']  
291826  
>>> countyData['AK']['Anchorage']['tracts']  
55
```

More generally, the `countyData` dictionary's keys will look like this:

```
countyData[state abbrev][county]['tracts']  
countyData[state abbrev][county]['pop']
```

Now that you know how `countyData` will be structured, you can write the code that will fill it with the county data. Add the following code to the bottom of your program:

```
#! python 3  
# readCensusExcel.py - Tabulates population and number of census tracts for  
# each county.
```

```
--snip--
```

```

for row in range(2, sheet.get_highest_row() + 1):
    # Each row in the spreadsheet has data for one census tract.
    state = sheet['B' + str(row)].value
    county = sheet['C' + str(row)].value
    pop = sheet['D' + str(row)].value

    # Make sure the key for this state exists.
❶ countyData.setdefault(state, {})
    # Make sure the key for this county in this state exists.
❷ countyData[state].setdefault(county, {'tracts': 0, 'pop': 0})

    # Each row represents one census tract, so increment by one.
❸ countyData[state][county]['tracts'] += 1
    # Increase the county pop by the pop in this census tract.
❹ countyData[state][county]['pop'] += int(pop)

# TODO: Open a new text file and write the contents of countyData to it.

```

The last two lines of code perform the actual calculation work, incrementing the value for tracts ❸ and increasing the value for pop ❹ for the current county on each iteration of the for loop.

The other code is there because you cannot add a county dictionary as the value for a state abbreviation key until the key itself exists in countyData. (That is, countyData['AK']['Anchorage']['tracts'] += 1 will cause an error if the 'AK' key doesn't exist yet.) To make sure the state abbreviation key exists in your data structure, you need to call the setdefault() method to set a value if one does not already exist for state ❶.

Just as the countyData dictionary needs a dictionary as the value for each state abbreviation key, each of *those* dictionaries will need its own dictionary as the value for each county key ❷. And each of *those* dictionaries in turn will need keys 'tracts' and 'pop' that start with the integer value 0. (If you ever lose track of the dictionary structure, look back at the example dictionary at the start of this section.)

Since setdefault() will do nothing if the key already exists, you can call it on every iteration of the for loop without a problem.

Step 3: Write the Results to a File

After the for loop has finished, the countyData dictionary will contain all of the population and tract information keyed by county and state. At this point, you could program more code to write this to a text file or another Excel spreadsheet. For now, let's just use the pprint.pformat() function to write the countyData dictionary value as a massive string to a file named *census2010.py*. Add the following code to the bottom of your program (making sure to keep it unindented so that it stays outside the for loop):

```

#! python 3
# readCensusExcel.py - Tabulates population and number of census tracts for
# each county.

```

```
--snip--  
  
for row in range(2, sheet.get_highest_row() + 1):  
--snip--  
  
# Open a new text file and write the contents of countyData to it.  
print('Writing results...')  
resultFile = open('census2010.py', 'w')  
resultFile.write('allData = ' + pprint.pformat(countyData))  
resultFile.close()  
print('Done.')
```

The `pprint.pformat()` function produces a string that itself is formatted as valid Python code. By outputting it to a text file named `census2010.py`, you've generated a Python program from your Python program! This may seem complicated, but the advantage is that you can now import `census2010.py` just like any other Python module. In the interactive shell, change the current working directory to the folder with your newly created `census2010.py` file (on my laptop, this is `C:\Python34`), and then import it:

```
>>> import os  
>>> os.chdir('C:\\Python34')  
>>> import census2010  
>>> census2010.allData['AK']['Anchorage']  
{'pop': 291826, 'tracts': 55}  
>>> anchoragePop = census2010.allData['AK']['Anchorage']['pop']  
>>> print('The 2010 population of Anchorage was ' + str(anchoragePop))  
The 2010 population of Anchorage was 291826
```

The `readCensusExcel.py` program was throwaway code: Once you have its results saved to `census2010.py`, you won't need to run the program again. Whenever you need the county data, you can just run `import census2010`.

Calculating this data by hand would have taken hours; this program did it in a few seconds. Using OpenPyXL, you will have no trouble extracting information that is saved to an Excel spreadsheet and performing calculations on it. You can download the complete program from <http://nostarch.com/automatestuff/>.

Ideas for Similar Programs

Many businesses and offices use Excel to store various types of data, and it's not uncommon for spreadsheets to become large and unwieldy. Any program that parses an Excel spreadsheet has a similar structure: It loads the spreadsheet file, preps some variables or data structures, and then loops through each of the rows in the spreadsheet. Such a program could do the following:

- Compare data across multiple rows in a spreadsheet.
- Open multiple Excel files and compare data between spreadsheets.

- Check whether a spreadsheet has blank rows or invalid data in any cells and alert the user if it does.
- Read data from a spreadsheet and use it as the input for your Python programs.

Writing Excel Documents

OpenPyXL also provides ways of writing data, meaning that your programs can create and edit spreadsheet files. With Python, it's simple to create spreadsheets with thousands of rows of data.

Creating and Saving Excel Documents

Call the `openpyxl.Workbook()` function to create a new, blank `Workbook` object. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.get_sheet_names()
['Sheet']
>>> sheet = wb.get_active_sheet()
>>> sheet.title
'Sheet'
>>> sheet.title = 'Spam Bacon Eggs Sheet'
>>> wb.get_sheet_names()
['Spam Bacon Eggs Sheet']
```

The workbook will start off with a single sheet named `Sheet`. You can change the name of the sheet by storing a new string in its `title` attribute.

Any time you modify the `Workbook` object or its sheets and cells, the spreadsheet file will not be saved until you call the `save()` workbook method. Enter the following into the interactive shell (with `example.xlsx` in the current working directory):

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_active_sheet()
>>> sheet.title = 'Spam Spam Spam'
>>> wb.save('example_copy.xlsx')
```

Here, we change the name of our sheet. To save our changes, we pass a filename as a string to the `save()` method. Passing a different filename than the original, such as '`example_copy.xlsx`', saves the changes to a copy of the spreadsheet.

Whenever you edit a spreadsheet you've loaded from a file, you should always save the new, edited spreadsheet to a different filename than the original. That way, you'll still have the original spreadsheet file to work with in case a bug in your code caused the new, saved file to have incorrect or corrupt data.

Creating and Removing Sheets

Sheets can be added to and removed from a workbook with the `create_sheet()` and `remove_sheet()` methods. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.get_sheet_names()
['Sheet']
>>> wb.create_sheet()
<Worksheet "Sheet1">
>>> wb.get_sheet_names()
['Sheet', 'Sheet1']
>>> wb.create_sheet(index=0, title='First Sheet')
<Worksheet "First Sheet">
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Sheet1']
>>> wb.create_sheet(index=2, title='Middle Sheet')
<Worksheet "Middle Sheet">
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
```

The `create_sheet()` method returns a new `Worksheet` object named `SheetX`, which by default is set to be the last sheet in the workbook. Optionally, the index and name of the new sheet can be specified with the `index` and `title` keyword arguments.

Continue the previous example by entering the following:

```
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
>>> wb.remove_sheet(wb.get_sheet_by_name('Middle Sheet'))
>>> wb.remove_sheet(wb.get_sheet_by_name('Sheet1'))
>>> wb.get_sheet_names()
['First Sheet', 'Sheet']
```

The `remove_sheet()` method takes a `Worksheet` object, not a string of the sheet name, as its argument. If you know only the name of a sheet you want to remove, call `get_sheet_by_name()` and pass its return value into `remove_sheet()`.

Remember to call the `save()` method to save the changes after adding sheets to or removing sheets from the workbook.

Writing Values to Cells

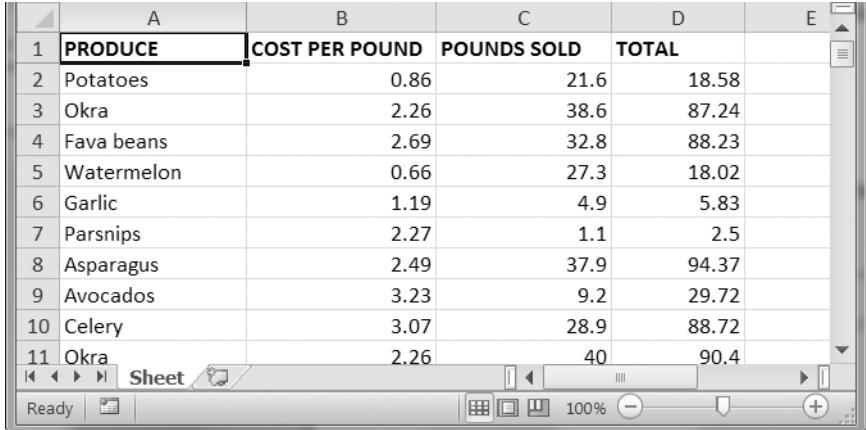
Writing values to cells is much like writing values to keys in a dictionary. Enter this into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_sheet_by_name('Sheet')
>>> sheet['A1'] = 'Hello world!'
>>> sheet['A1'].value
'Hello world!'
```

If you have the cell's coordinate as a string, you can use it just like a dictionary key on the `Worksheet` object to specify which cell to write to.

Project: Updating a Spreadsheet

In this project, you'll write a program to update cells in a spreadsheet of produce sales. Your program will look through the spreadsheet, find specific kinds of produce, and update their prices. Download this spreadsheet from <http://nostarch.com/automatestuff/>. Figure 12-3 shows what the spreadsheet looks like.



The screenshot shows a Microsoft Excel spreadsheet titled "Sheet1". The data is organized into columns A through E. Column A is labeled "PRODUCE", column B is "COST PER POUND", column C is "POUNDS SOLD", and column D is "TOTAL". The data rows are numbered 1 through 11. Row 1 contains the headers. Rows 2 through 10 list individual produce items with their respective cost per pound and pounds sold. Row 11 lists "Okra" again with a different cost per pound and total revenue. The bottom of the screen shows the standard Excel ribbon tabs (File, Home, Insert, etc.) and a status bar indicating "Ready" and "100%".

	A	B	C	D	E
1	PRODUCE	COST PER POUND	POUNDS SOLD	TOTAL	
2	Potatoes	0.86	21.6	18.58	
3	Okra	2.26	38.6	87.24	
4	Fava beans	2.69	32.8	88.23	
5	Watermelon	0.66	27.3	18.02	
6	Garlic	1.19	4.9	5.83	
7	Parsnips	2.27	1.1	2.5	
8	Asparagus	2.49	37.9	94.37	
9	Avocados	3.23	9.2	29.72	
10	Celery	3.07	28.9	88.72	
11	Okra	2.26	40	90.4	

Figure 12-3: A spreadsheet of produce sales

Each row represents an individual sale. The columns are the type of produce sold (A), the cost per pound of that produce (B), the number of pounds sold (C), and the total revenue from the sale (D). The TOTAL column is set to the Excel formula `=ROUND(B3*C3, 2)`, which multiplies the cost per pound by the number of pounds sold and rounds the result to the nearest cent. With this formula, the cells in the TOTAL column will automatically update themselves if there is a change in column B or C.

Now imagine that the prices of garlic, celery, and lemons were entered incorrectly, leaving you with the boring task of going through thousands of rows in this spreadsheet to update the cost per pound for any garlic, celery, and lemon rows. You can't do a simple find-and-replace for the price because there might be other items with the same price that you don't want to mistakenly "correct." For thousands of rows, this would take hours to do by hand. But you can write a program that can accomplish this in seconds.

Your program does the following:

- Loops over all the rows.
- If the row is for garlic, celery, or lemons, changes the price.

This means your code will need to do the following:

- Open the spreadsheet file.
- For each row, check whether the value in column A is Celery, Garlic, or Lemon.
- If it is, update the price in column B.
- Save the spreadsheet to a new file (so that you don't lose the old spreadsheet, just in case).

Step 1: Set Up a Data Structure with the Update Information

The prices that you need to update are as follows:

Celery	1.19
Garlic	3.07
Lemon	1.27

You could write code like this:

```
if produceName == 'Celery':  
    cell0bj = 1.19  
if produceName == 'Garlic':  
    cell0bj = 3.07  
if produceName == 'Lemon':  
    cell0bj = 1.27
```

Having the produce and updated price data hardcoded like this is a bit inelegant. If you needed to update the spreadsheet again with different prices or different produce, you would have to change a lot of the code. Every time you change code, you risk introducing bugs.

A more flexible solution is to store the corrected price information in a dictionary and write your code to use this data structure. In a new file editor window, enter the following code:

```
#! python3  
# updateProduce.py - Corrects costs in produce sales spreadsheet.  
  
import openpyxl  
  
wb = openpyxl.load_workbook('produceSales.xlsx')  
sheet = wb.get_sheet_by_name('Sheet')  
  
# The produce types and their updated prices  
PRICE_UPDATES = {'Garlic': 3.07,  
                 'Celery': 1.19,  
                 'Lemon': 1.27}  
  
# TODO: Loop through the rows and update the prices.
```

Save this as *updateProduce.py*. If you need to update the spreadsheet again, you'll need to update only the PRICE_UPDATES dictionary, not any other code.

Step 2: Check All Rows and Update Incorrect Prices

The next part of the program will loop through all the rows in the spreadsheet. Add the following code to the bottom of *updateProduce.py*:

```
#! python3
# updateProduce.py - Corrects costs in produce sales spreadsheet.

--snip--

# Loop through the rows and update the prices.
❶ for rowNum in range(2, sheet.get_highest_row()):    # skip the first row
❷     produceName = sheet.cell(row=rowNum, column=1).value
❸     if produceName in PRICE_UPDATES:
        sheet.cell(row=rowNum, column=2).value = PRICE_UPDATES[produceName]

❹ wb.save('updatedProduceSales.xlsx')
```

We loop through the rows starting at row 2, since row 1 is just the header ❶. The cell in column 1 (that is, column A) will be stored in the variable `produceName` ❷. If `produceName` exists as a key in the `PRICE_UPDATES` dictionary ❸, then you know this is a row that must have its price corrected. The correct price will be in `PRICE_UPDATES[produceName]`.

Notice how clean using `PRICE_UPDATES` makes the code. Only one `if` statement, rather than code like `if produceName == 'Garlic':`, is necessary for every type of produce to update. And since the code uses the `PRICE_UPDATES` dictionary instead of hardcoding the produce names and updated costs into the `for` loop, you modify only the `PRICE_UPDATES` dictionary and not the code if the produce sales spreadsheet needs additional changes.

After going through the entire spreadsheet and making changes, the code saves the `Workbook` object to *updatedProduceSales.xlsx* ❹. It doesn't overwrite the old spreadsheet just in case there's a bug in your program and the updated spreadsheet is wrong. After checking that the updated spreadsheet looks right, you can delete the old spreadsheet.

You can download the complete source code for this program from <http://nostarch.com/automatestuff/>.

Ideas for Similar Programs

Since many office workers use Excel spreadsheets all the time, a program that can automatically edit and write Excel files could be really useful. Such a program could do the following:

- Read data from one spreadsheet and write it to parts of other spreadsheets.
- Read data from websites, text files, or the clipboard and write it to a spreadsheet.
- Automatically “clean up” data in spreadsheets. For example, it could use regular expressions to read multiple formats of phone numbers and edit them to a single, standard format.

Setting the Font Style of Cells

Styling certain cells, rows, or columns can help you emphasize important areas in your spreadsheet. In the produce spreadsheet, for example, your program could apply bold text to the potato, garlic, and parsnip rows. Or perhaps you want to italicize every row with a cost per pound greater than \$5. Styling parts of a large spreadsheet by hand would be tedious, but your programs can do it instantly.

To customize font styles in cells, important, import the `Font()` and `Style()` functions from the `openpyxl.styles` module.

```
from openpyxl.styles import Font, Style
```

This allows you to type `Font()` instead of `openpyxl.styles.Font()`. (See “Importing Modules” on page 57 to review this style of `import` statement.)

Here’s an example that creates a new workbook and sets cell A1 to have a 24-point, italicized font. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> from openpyxl.styles import Font, Style
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_sheet_by_name('Sheet')
❶ >>> italic24Font = Font(size=24, italic=True)
❷ >>> styleObj = Style(font=italic24Font)
❸ >>> sheet['A'].style=styleObj
>>> sheet['A1'] = 'Hello world!'
>>> wb.save('styled.xlsx')
```

OpenPyXL represents the collection of style settings for a cell with a `Style` object, which is stored in the `Cell` object’s `style` attribute. A cell’s style can be set by assigning the `Style` object to the `style` attribute.

In this example, `Font(size=24, italic=True)` returns a `Font` object, which is stored in `italic24Font` ❶. The keyword arguments to `Font()`, `size` and `italic`, configure the `Font` object’s style attributes. This `Font` object is then passed into the `Style(font=italic24Font)` call, which returns the value you stored in `styleObj` ❷. And when `styleObj` is assigned to the cell’s `style` attribute ❸, all that font styling information gets applied to cell A1.

Font Objects

The `style` attributes in `Font` objects affect how the text in cells is displayed. To set font style attributes, you pass keyword arguments to `Font()`. Table 12-2 shows the possible keyword arguments for the `Font()` function.

Table 12-2: Keyword Arguments for Font style Attributes

Keyword argument	Data type	Description
name	String	The font name, such as 'Calibri' or 'Times New Roman'
size	Integer	The point size
bold	Boolean	True, for bold font
italic	Boolean	True, for italic font

You can call `Font()` to create a `Font` object and store that `Font` object in a variable. You then pass that to `Style()`, store the resulting `Style` object in a variable, and assign that variable to a `Cell` object's `style` attribute. For example, this code creates various font styles:

```
>>> import openpyxl
>>> from openpyxl.styles import Font, Style
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_sheet_by_name('Sheet')

>>> fontObj1 = Font(name='Times New Roman', bold=True)
>>> styleObj1 = Style(font=fontObj1)
>>> sheet['A1'].style=styleObj1
>>> sheet['A1'] = 'Bold Times New Roman'

>>> fontObj2 = Font(size=24, italic=True)
>>> styleObj2 = Style(font=fontObj2)
>>> sheet['B3'].style=styleObj2
>>> sheet['B3'] = '24 pt Italic'

>>> wb.save('styles.xlsx')
```

Here, we store a `Font` object in `fontObj1` and use it to create a `Style` object, which we store in `styleObj1`, and then set the `A1` `Cell` object's `style` attribute to `styleObj1`. We repeat the process with another `Font` object and `Style` object to set the style of a second cell. After you run this code, the styles of the `A1` and `B3` cells in the spreadsheet will be set to custom font styles, as shown in Figure 12-4.

	A	B	C	D
1	Bold Times New Roman			
2				
3		<i>24 pt Italic</i>		
4				
5				

Figure 12-4: A spreadsheet with custom font styles

For cell A1, we set the font name to 'Times New Roman' and set bold to true, so our text appears in bold Times New Roman. We didn't specify a size, so the openpyxl default, 11, is used. In cell B3, our text is italic, with a size of 24; we didn't specify a font name, so the openpyxl default, Calibri, is used.

Formulas

Formulas, which begin with an equal sign, can configure cells to contain values calculated from other cells. In this section, you'll use the openpyxl module to programmatically add formulas to cells, just like any normal value. For example:

```
>>> sheet['B9'] = '=SUM(B1:B8)'
```

This will store $=SUM(B1:B8)$ as the value in cell B9. This sets the B9 cell to a formula that calculates the sum of values in cells B1 to B8. You can see this in action in Figure 12-5.



Figure 12-5: Cell B9 contains the formula $=SUM(B1:B8)$, which adds the cells B1 to B8.

A formula is set just like any other text value in a cell. Enter the following into the interactive shell:

```
>>> import openpyxl  
>>> wb = openpyxl.Workbook()  
>>> sheet = wb.get_active_sheet()  
>>> sheet['A1'] = 200  
>>> sheet['A2'] = 300  
>>> sheet['A3'] = '=SUM(A1:A2)'  
>>> wb.save('writeFormula.xlsx')
```

The cells in A1 and A2 are set to 200 and 300, respectively. The value in cell A3 is set to a formula that sums the values in A1 and A2. When the spreadsheet is opened in Excel, A3 will display its value as 500.

You can also read the formula in a cell just as you would any value. However, if you want to see the result of the *calculation* for the formula instead of the literal formula, you must pass `True` for the `data_only` keyword argument to `load_workbook()`. This means a `Workbook` object can show either the formulas or the result of the formulas but not both. (But you can have multiple `Workbook` objects loaded for the same spreadsheet file.) Enter the following into the interactive shell to see the difference between loading a workbook with and without the `data_only` keyword argument:

```
>>> import openpyxl  
>>> wbFormulas = openpyxl.load_workbook('writeFormula.xlsx')  
>>> sheet = wbFormulas.get_active_sheet()  
>>> sheet['A3'].value  
'=SUM(A1:A2)'  
  
>>> wbDataOnly = openpyxl.load_workbook('writeFormula.xlsx', data_only=True)  
>>> sheet = wbDataOnly.get_active_sheet()  
>>> sheet['A3'].value  
500
```

Here, when `load_workbook()` is called with `data_only=True`, the A3 cell shows 500, the result of the `=SUM(A1:A2)` formula, rather than the text of the formula.

Excel formulas offer a level of programmability for spreadsheets but can quickly become unmanageable for complicated tasks. For example, even if you're deeply familiar with Excel formulas, it's a headache to try to decipher what `=IFERROR(TRIM(IF(LEN(VLOOKUP(F7, Sheet2!A1:B10000, 2,
FALSE))>0,SUBSTITUTE(VLOOKUP(F7, Sheet2!A1:B10000, 2, FALSE),
" ", ""), "")), "")` actually does. Python code is much more readable.

Adjusting Rows and Columns

In Excel, adjusting the sizes of rows and columns is as easy as clicking and dragging the edges of a row or column header. But if you need to set a row or column's size based on its cells' contents or if you want to set sizes in a large number of spreadsheet files, it will be much quicker to write a Python program to do it.

Rows and columns can also be hidden entirely from view. Or they can be “frozen” in place so that they are always visible on the screen and appear on every page when the spreadsheet is printed (which is handy for headers).

Setting Row Height and Column Width

Worksheet objects have `row_dimensions` and `column_dimensions` attributes that control row heights and column widths. Enter this into the interactive shell:

```
>>> import openpyxl  
>>> wb = openpyxl.Workbook()  
>>> sheet = wb.get_active_sheet()  
>>> sheet['A1'] = 'Tall row'  
>>> sheet['B2'] = 'Wide column'  
>>> sheet.row_dimensions[1].height = 70  
>>> sheet.column_dimensions['B'].width = 20  
>>> wb.save('dimensions.xlsx')
```

A sheet's `row_dimensions` and `column_dimensions` are dictionary-like values; `row_dimensions` contains `RowDimension` objects and `column_dimensions` contains `ColumnDimension` objects. In `row_dimensions`, you can access one of the objects using the number of the row (in this case, 1 or 2). In `column_dimensions`, you can access one of the objects using the letter of the column (in this case, A or B).

The `dimensions.xlsx` spreadsheet looks like Figure 12-6.

	A	B
1	Tall row	
2		Wide column
3		

Figure 12-6: Row 1 and column B set to larger heights and widths

Once you have the `RowDimension` object, you can set its height. Once you have the `ColumnDimension` object, you can set its width. The row height can be set to an integer or float value between 0 and 409. This value represents the height measured in *points*, where one point equals 1/72 of an inch. The default row height is 12.75. The column width can be set to an integer or float value between 0 and 255. This value represents the number of characters at the default font size (11 point) that can be displayed in the cell. The default column width is 8.43 characters. Columns with widths of 0 or rows with heights of 0 are hidden from the user.

Merging and Unmerging Cells

A rectangular area of cells can be merged into a single cell with the `merge_cells()` sheet method. Enter the following into the interactive shell:

```
>>> import openpyxl  
>>> wb = openpyxl.Workbook()
```

```
>>> sheet = wb.get_active_sheet()
>>> sheet.merge_cells('A1:D3')
>>> sheet['A1'] = 'Twelve cells merged together.'
>>> sheet.merge_cells('C5:D5')
>>> sheet['C5'] = 'Two merged cells.'
>>> wb.save('merged.xlsx')
```

The argument to `merge_cells()` is a single string of the top-left and bottom-right cells of the rectangular area to be merged: '`A1:D3`' merges 12 cells into a single cell. To set the value of these merged cells, simply set the value of the top-left cell of the merged group.

When you run this code, `merged.xlsx` will look like Figure 12-7.

	A	B	C	D	E
1					
2					
3	Twelve cells merged together.				
4					
5			Two merged cells.		
6					
7					

Figure 12-7: Merged cells in a spreadsheet

To unmerge cells, call the `unmerge_cells()` sheet method. Enter this into the interactive shell.

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('merged.xlsx')
>>> sheet = wb.get_active_sheet()
>>> sheet.unmerge_cells('A1:D3')
>>> sheet.unmerge_cells('C5:D5')
>>> wb.save('merged.xlsx')
```

If you save your changes and then take a look at the spreadsheet, you'll see that the merged cells have gone back to being individual cells.

Freeze Panes

For spreadsheets too large to be displayed all at once, it's helpful to "freeze" a few of the top rows or leftmost columns onscreen. Frozen column or row headers, for example, are always visible to the user even as they scroll through the spreadsheet. These are known as *freeze panes*. In OpenPyXL, each `Worksheet` object has a `freeze_panes` attribute that can be set to a `Cell` object or a string of a cell's coordinates. Note that all rows above and all columns to the left of this cell will be frozen, but the row and column of the cell itself will not be frozen.

To unfreeze all panes, set `freeze_panes` to `None` or '`A1`'. Table 12-3 shows which rows and columns will be frozen for some example settings of `freeze_panes`.

Table 12-3: Frozen Pane Examples

freeze_panes setting	Rows and columns frozen
sheet.freeze_panes = 'A2'	Row 1
sheet.freeze_panes = 'B1'	Column A
sheet.freeze_panes = 'C1'	Columns A and B
sheet.freeze_panes = 'C2'	Row 1 and columns A and B
sheet.freeze_panes = 'A1' or sheet.freeze_panes = None	No frozen panes

Make sure you have the produce sales spreadsheet from <http://nostarch.com/automatestuff/>. Then enter the following into the interactive shell:

```
>>> import openpyxl  
>>> wb = openpyxl.load_workbook('produceSales.xlsx')  
>>> sheet = wb.get_active_sheet()  
>>> sheet.freeze_panes = 'A2'  
>>> wb.save('freezeExample.xlsx')
```

If you set the `freeze_panes` attribute to '`A2`', row 1 will always be viewable, no matter where the user scrolls in the spreadsheet. You can see this in Figure 12-8.



Figure 12-8: With `freeze_panes` set to '`A2`', row 1 is always visible even as the user scrolls down.

Charts

OpenPyXL supports creating bar, line, scatter, and pie charts using the data in a sheet's cells. To make a chart, you need to do the following:

1. Create a `Reference` object from a rectangular selection of cells.
2. Create a `Series` object by passing in the `Reference` object.
3. Create a `Chart` object.

4. Append the Series object to the Chart object.
5. Optionally, set the `drawing.top`, `drawing.left`, `drawing.width`, and `drawing.height` variables of the Chart object.
6. Add the Chart object to the Worksheet object.

The Reference object requires some explaining. Reference objects are created by calling the `openpyxl.charts.Reference()` function and passing three arguments:

1. The Worksheet object containing your chart data.
2. A tuple of two integers, representing the top-left cell of the rectangular selection of cells containing your chart data: The first integer in the tuple is the row, and the second is the column. Note that 1 is the first row, not 0.
3. A tuple of two integers, representing the bottom-right cell of the rectangular selection of cells containing your chart data: The first integer in the tuple is the row, and the second is the column.

Figure 12-9 shows some sample coordinate arguments.

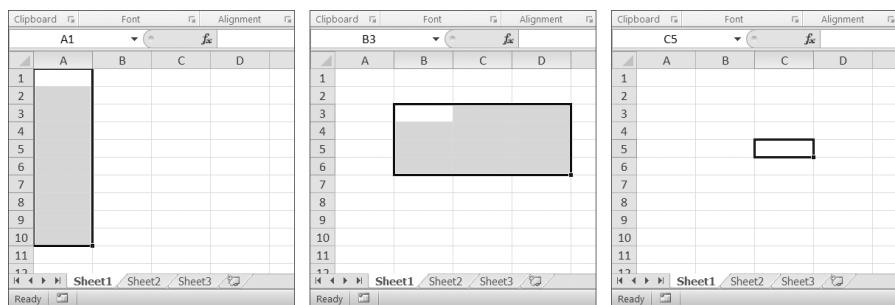


Figure 12-9: From left to right: (1, 1), (10, 1); (3, 2), (6, 4); (5, 3), (5, 3)

Enter this interactive shell example to create a bar chart and add it to the spreadsheet:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_active_sheet()
>>> for i in range(1, 11):          # create some data in column A
    sheet['A' + str(i)] = i

>>> refObj = openpyxl.charts.Reference(sheet, (1, 1), (10, 1))

>>> seriesObj = openpyxl.charts.Series(refObj, title='First series')

>>> chartObj = openpyxl.charts.BarChart()
>>> chartObj.append(seriesObj)
```

```
>>> chartObj.drawing.top = 50      # set the position
>>> chartObj.drawing.left = 100
>>> chartObj.drawing.width = 300    # set the size
>>> chartObj.drawing.height = 200

>>> sheet.add_chart(chartObj)
>>> wb.save('sampleChart.xlsx')
```

This produces a spreadsheet that looks like Figure 12-10.



Figure 12-10: A spreadsheet with a chart added

We've created a bar chart by calling `openpyxl.charts.BarChart()`. You can also create line charts, scatter charts, and pie charts by calling `openpyxl.charts.LineChart()`, `openpyxl.charts.ScatterChart()`, and `openpyxl.charts.PieChart()`.

Unfortunately, in the current version of OpenPyXL (2.1.4), the `load_workbook()` function does not load charts in Excel files. Even if the Excel file has charts, the loaded Workbook object will not include them. If you load a Workbook object and immediately save it to the same `.xlsx` filename, you will effectively remove the charts from it.

Summary

Often the hard part of processing information isn't the processing itself but simply getting the data in the right format for your program. But once you have your spreadsheet loaded into Python, you can extract and manipulate its data much faster than you could by hand.

You can also generate spreadsheets as output from your programs. So if colleagues need your text file or PDF of thousands of sales contacts transferred to a spreadsheet file, you won't have to tediously copy and paste it all into Excel.

Equipped with the `openpyxl` module and some programming knowledge, you'll find processing even the biggest spreadsheets a piece of cake.

Practice Questions

For the following questions, imagine you have a `Workbook` object in the variable `wb`, a `Worksheet` object in `sheet`, a `Cell` object in `cell`, a `Comment` object in `comm`, and an `Image` object in `img`.

1. What does the `openpyxl.load_workbook()` function return?
2. What does the `get_sheet_names()` workbook method return?
3. How would you retrieve the `Worksheet` object for a sheet named 'Sheet1'?
4. How would you retrieve the `Worksheet` object for the workbook's active sheet?
5. How would you retrieve the value in the cell C5?
6. How would you set the value in the cell C5 to "Hello"?
7. How would you retrieve the cell's row and column as integers?
8. What do the `get_highest_column()` and `get_highest_row()` sheet methods return, and what is the data type of these return values?
9. If you needed to get the integer index for column 'M', what function would you need to call?
10. If you needed to get the string name for column 14, what function would you need to call?
11. How can you retrieve a tuple of all the `Cell` objects from A1 to F1?
12. How would you save the workbook to the filename `example.xlsx`?
13. How do you set a formula in a cell?
14. If you want to retrieve the result of a cell's formula instead of the cell's formula itself, what must you do first?
15. How would you set the height of row 5 to 100?
16. How would you hide column C?
17. Name a few features that OpenPyXL 2.1.4 does not load from a spreadsheet file.
18. What is a freeze pane?
19. What five functions and methods do you have to call to create a bar chart?

Practice Projects

For practice, write programs that perform the following tasks.

Multiplication Table Maker

Create a program `multiplicationTable.py` that takes a number N from the command line and creates an $N \times N$ multiplication table in an Excel spreadsheet. For example, when the program is run like this:

```
py multiplicationTable.py 6
```

... it should create a spreadsheet that looks like Figure 12-11.

Figure 12-11: A multiplication table generated in a spreadsheet

Row 1 and column A should be used for labels and should be in bold.

Blank Row Inserter

Create a program `blankRowInserter.py` that takes two integers and a filename string as command line arguments. Let's call the first integer N and the second integer M . Starting at row N , the program should insert M blank rows into the spreadsheet. For example, when the program is run like this:

```
python blankRowInserter.py 3 2 myProduce.xlsx
```

... the “before” and “after” spreadsheets should look like Figure 12-12.

Figure 12-12: Before (left) and after (right) the two blank rows are inserted at row 3

You can write this program by reading in the contents of the spreadsheet. Then, when writing out the new spreadsheet, use a `for` loop to copy the first N lines. For the remaining lines, add M to the row number in the output spreadsheet.

Spreadsheet Cell Inverter

Write a program to invert the row and column of the cells in the spreadsheet. For example, the value at row 5, column 3 will be at row 3, column 5 (and vice versa). This should be done for all cells in the spreadsheet. For example, the “before” and “after” spreadsheets would look something like Figure 12-13.

A	B	C	D	E	F	G	H	I	J
ITEM	SOLD								
2	Eggplant	334							
3	Cucumber	252							
4	Green cabl	238							
5	Eggplant	516							
6	Garlic	98							
7	Parsnips	16							
8	Asparagus	335							
9	Avocados	84							
10									

A	B	C	D	E	F	G	H	I	J
ITEM	Eggplant	Cucumber	Green cabl	Eggplant	Garlic	Parsnips	Asparagus	Avocados	
2	SOLD	334	252	238	516	98	16	335	84
3									
4									
5									
6									
7									
8									
9									
10									

Figure 12-13: The spreadsheet before (top) and after (bottom) inversion

You can write this program by using nested `for` loops to read in the spreadsheet's data into a list of lists data structure. This data structure could have `sheetData[x][y]` for the cell at column x and row y. Then, when writing out the new spreadsheet, use `sheetData[y][x]` for the cell at column x and row y.

Text Files to Spreadsheet

Write a program to read in the contents of several text files (you can make the text files yourself) and insert those contents into a spreadsheet, with one line of text per row. The lines of the first text file will be in the cells of column A, the lines of the second text file will be in the cells of column B, and so on.

Use the `readlines()` File object method to return a list of strings, one string per line in the file. For the first file, output the first line to column 1, row 1. The second line should be written to column 1, row 2, and so on. The next file that is read with `readlines()` will be written to column 2, the next file to column 3, and so on.

Spreadsheet to Text Files

Write a program that performs the tasks of the previous program in reverse order: The program should open a spreadsheet and write the cells of column A into one text file, the cells of column B into another text file, and so on.

13

WORKING WITH PDF AND WORD DOCUMENTS



PDF and Word documents are binary files, which makes them much more complex than plaintext files. In addition to text, they store lots of font, color, and layout information. If you want your programs to read or write to PDFs or Word documents, you'll need to do more than simply pass their filenames to `open()`.

Fortunately, there are Python modules that make it easy for you to interact with PDFs and Word documents. This chapter will cover two such modules: PyPDF2 and Python-Docx.

PDF Documents

PDF stands for *Portable Document Format* and uses the `.pdf` file extension. Although PDFs support many features, this chapter will focus on the two things you'll be doing most often with them: reading text content from PDFs and crafting new PDFs from existing documents.

The module you'll use to work with PDFs is PyPDF2. To install it, run `pip install PyPDF2` from the command line. This module name is case sensitive, so make sure the `y` is lowercase and everything else is uppercase. (Check out Appendix A for full details about installing third-party modules.) If the module was installed correctly, running `import PyPDF2` in the interactive shell shouldn't display any errors.

THE PROBLEMATIC PDF FORMAT

While PDF files are great for laying out text in a way that's easy for people to print and read, they're not straightforward for software to parse into plaintext. As such, PyPDF2 might make mistakes when extracting text from a PDF and may even be unable to open some PDFs at all. There isn't much you can do about this, unfortunately. PyPDF2 may simply be unable to work with some of your particular PDF files. That said, I haven't found any PDF files so far that can't be opened with PyPDF2.

Extracting Text from PDFs

PyPDF2 does not have a way to extract images, charts, or other media from PDF documents, but it can extract text and return it as a Python string. To start learning how PyPDF2 works, we'll use it on the example PDF shown in Figure 13-1.

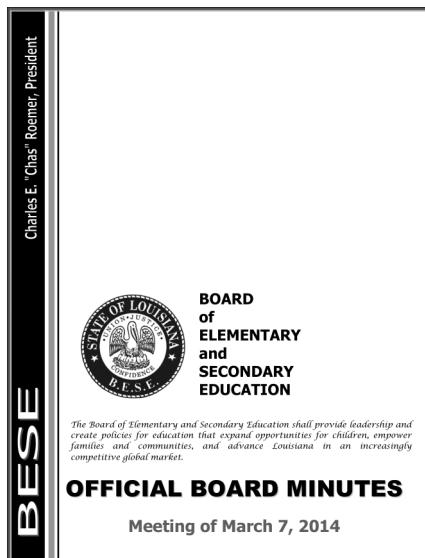


Figure 13-1: The PDF page that we will be extracting text from

Download this PDF from <http://nostarch.com/automatestuff/>, and enter the following into the interactive shell:

```
>>> import PyPDF2
>>> pdfFileObj = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
❶ >>> pdfReader.numPages
19
❷ >>> pageObj = pdfReader.getPage(0)
❸ >>> pageObj.extractText()
'OOFFFFIICCIAALL BBOOAARRDD MMIINNUUTTEESS Meeting of March 7, 2015
\n      The Board of Elementary and Secondary Education shall provide leadership
and create policies for education that expand opportunities for children,
empower families and communities, and advance Louisiana in an increasingly
competitive global market. BOARD of ELEMENTARY and SECONDARY EDUCATION '
```

First, import the `PyPDF2` module. Then open `meetingminutes.pdf` in read binary mode and store it in `pdfFileObj`. To get a `PdfFileReader` object that represents this PDF, call `PyPDF2.PdfFileReader()` and pass it `pdfFileObj`. Store this `PdfFileReader` object in `pdfReader`.

The total number of pages in the document is stored in the `numPages` attribute of a `PdfFileReader` object ❶. The example PDF has 19 pages, but let's extract text from only the first page.

To extract text from a page, you need to get a `Page` object, which represents a single page of a PDF, from a `PdfFileReader` object. You can get a `Page` object by calling the `getPage()` method ❷ on a `PdfFileReader` object and passing it the page number of the page you're interested in—in our case, 0.

`PyPDF2` uses a *zero-based index* for getting pages: The first page is page 0, the second is page 1, and so on. This is always the case, even if pages are numbered differently within the document. For example, say your PDF is a three-page excerpt from a longer report, and its pages are numbered 42, 43, and 44. To get the first page of this document, you would want to call `pdfReader.getPage(0)`, not `getPage(42)` or `getPage(1)`.

Once you have your `Page` object, call its `extractText()` method to return a string of the page's text ❸. The text extraction isn't perfect: The text *Charles E. "Chas" Roemer, President* from the PDF is absent from the string returned by `extractText()`, and the spacing is sometimes off. Still, this approximation of the PDF text content may be good enough for your program.

Decrypting PDFs

Some PDF documents have an encryption feature that will keep them from being read until whoever is opening the document provides a password. Enter the following into the interactive shell with the PDF you downloaded, which has been encrypted with the password *rosebud*:

```
>>> import PyPDF2
>>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))
❶ >>> pdfReader.isEncrypted
True
```

```
>>> pdfReader.getPage(0)
❷ Traceback (most recent call last):
  File "<pyshell#173>", line 1, in <module>
    pdfReader.getPage()
--snip--
  File "C:\Python34\lib\site-packages\PyPDF2\pdf.py", line 1173, in get0bject
    raise utils.PdfReadError("file has not been decrypted")
PyPDF2.utils.PdfReadError: file has not been decrypted
❸ >>> pdfReader.decrypt('rosebud')
1
>>> page0bj = pdfReader.getPage(0)
```

All PdfFileReader objects have an `isEncrypted` attribute that is `True` if the PDF is encrypted and `False` if it isn't ❶. Any attempt to call a function that reads the file before it has been decrypted with the correct password will result in an error ❷.

To read an encrypted PDF, call the `decrypt()` function and pass the password as a string ❸. After you call `decrypt()` with the correct password, you'll see that calling `getPage()` no longer causes an error. If given the wrong password, the `decrypt()` function will return `0` and `getPage()` will continue to fail. Note that the `decrypt()` method decrypts only the PdfFileReader object, not the actual PDF file. After your program terminates, the file on your hard drive remains encrypted. Your program will have to call `decrypt()` again the next time it is run.

Creating PDFs

PyPDF2's counterpart to PdfFileReader objects is PdfFileWriter objects, which can create new PDF files. But PyPDF2 cannot write arbitrary text to a PDF like Python can do with plaintext files. Instead, PyPDF2's PDF-writing capabilities are limited to copying pages from other PDFs, rotating pages, overlaying pages, and encrypting files.

PyPDF2 doesn't allow you to directly edit a PDF. Instead, you have to create a new PDF and then copy content over from an existing document. The examples in this section will follow this general approach:

1. Open one or more existing PDFs (the source PDFs) into PdfFileReader objects.
2. Create a new PdfFileWriter object.
3. Copy pages from the PdfFileReader objects into the PdfFileWriter object.
4. Finally, use the PdfFileWriter object to write the output PDF.

Creating a PdfFileWriter object creates only a value that represents a PDF document in Python. It doesn't create the actual PDF file. For that, you must call the PdfFileWriter's `write()` method.

The `write()` method takes a regular `File` object that has been opened in *write-binary* mode. You can get such a `File` object by calling Python's `open()` function with two arguments: the string of what you want the PDF's filename to be and '`wb`' to indicate the file should be opened in write-binary mode.

If this sounds a little confusing, don't worry—you'll see how this works in the following code examples.

Copying Pages

You can use PyPDF2 to copy pages from one PDF document to another. This allows you to combine multiple PDF files, cut unwanted pages, or reorder pages.

Download `meetingminutes.pdf` and `meetingminutes2.pdf` from <http://nostarch.com/automatestuff/> and place the PDFs in the current working directory. Enter the following into the interactive shell:

```
>>> import PyPDF2
>>> pdf1File = open('meetingminutes.pdf', 'rb')
>>> pdf2File = open('meetingminutes2.pdf', 'rb')
❶ >>> pdf1Reader = PyPDF2.PdfFileReader(pdf1File)
❷ >>> pdf2Reader = PyPDF2.PdfFileReader(pdf2File)
❸ >>> pdfWriter = PyPDF2.PdfFileWriter()

>>> for pageNum in range(pdf1Reader.numPages):
❹     pageObj = pdf1Reader.getPage(pageNum)
❺     pdfWriter.addPage(pageObj)

>>> for pageNum in range(pdf2Reader.numPages):
❻     pageObj = pdf2Reader.getPage(pageNum)
❼     pdfWriter.addPage(pageObj)

❾ >>> pdfOutputFile = open('combinedminutes.pdf', 'wb')
>>> pdfWriter.write(pdfOutputFile)
>>> pdfOutputFile.close()
>>> pdf1File.close()
>>> pdf2File.close()
```

Open both PDF files in read binary mode and store the two resulting `File` objects in `pdf1File` and `pdf2File`. Call `PyPDF2.PdfFileReader()` and pass it `pdf1File` to get a `PdfFileReader` object for `meetingminutes.pdf` ❶. Call it again and pass it `pdf2File` to get a `PdfFileReader` object for `meetingminutes2.pdf` ❷. Then create a new `PdfFileWriter` object, which represents a blank PDF document ❸.

Next, copy all the pages from the two source PDFs and add them to the `PdfFileWriter` object. Get the `Page` object by calling `getPage()` on a `PdfFileReader` object ❹. Then pass that `Page` object to your `PdfFileWriter`'s `addPage()` method ❺. These steps are done first for `pdf1Reader` and then

again for `pdf2Reader`. When you’re done copying pages, write a new PDF called `combinedminutes.pdf` by passing a `File` object to the `PdfFileWriter`’s `write()` method ❻.

NOTE *PyPDF2 cannot insert pages in the middle of a PdfFileWriter object; the addPage() method will only add pages to the end.*

You have now created a new PDF file that combines the pages from `meetingminutes.pdf` and `meetingminutes2.pdf` into a single document. Remember that the `File` object passed to `PyPDF2.PdfFileReader()` needs to be opened in read-binary mode by passing `'rb'` as the second argument to `open()`. Likewise, the `File` object passed to `PyPDF2.PdfFileWriter()` needs to be opened in write-binary mode with `'wb'`.

Rotating Pages

The pages of a PDF can also be rotated in 90-degree increments with the `rotateClockwise()` and `rotateCounterClockwise()` methods. Pass one of the integers 90, 180, or 270 to these methods. Enter the following into the interactive shell, with the `meetingminutes.pdf` file in the current working directory:

```
>>> import PyPDF2
>>> minutesFile = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
❶ >>> page = pdfReader.getPage(0)
❷ >>> page.rotateClockwise(90)
{'/Contents': [IndirectObject(961, 0), IndirectObject(962, 0),
--snip--
]}
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> pdfWriter.addPage(page)
❸ >>> resultPdfFile = open('rotatedPage.pdf', 'wb')
>>> pdfWriter.write(resultPdfFile)
>>> resultPdfFile.close()
>>> minutesFile.close()
```

Here we use `getPage(0)` to select the first page of the PDF ❶, and then we call `rotateClockwise(90)` on that page ❷. We write a new PDF with the rotated page and save it as `rotatedPage.pdf` ❸.

The resulting PDF will have one page, rotated 90 degrees clockwise, as in Figure 13-2. The return values from `rotateClockwise()` and `rotateCounterClockwise()` contain a lot of information that you can ignore.

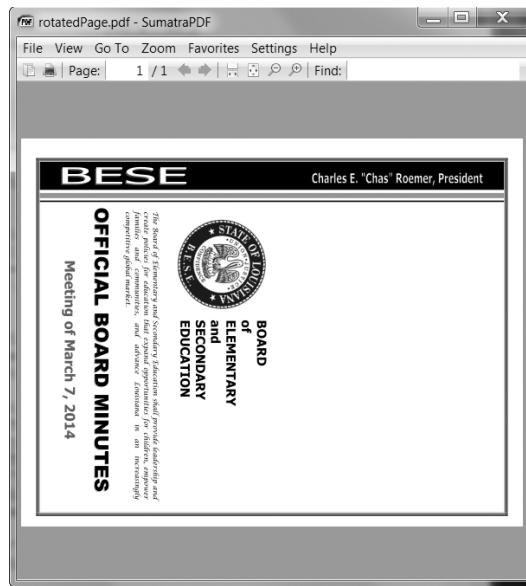


Figure 13-2: The rotatedPage.pdf file with the page rotated 90 degrees clockwise

Overlaying Pages

PyPDF2 can also overlay the contents of one page over another, which is useful for adding a logo, timestamp, or watermark to a page. With Python, it's easy to add watermarks to multiple files and only to pages your program specifies.

Download *watermark.pdf* from <http://nostarch.com/automatestuff/> and place the PDF in the current working directory along with *meetingminutes.pdf*. Then enter the following into the interactive shell:

```
>>> import PyPDF2
>>> minutesFile = open('meetingminutes.pdf', 'rb')
❶ >>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
❷ >>> minutesFirstPage = pdfReader.getPage(0)
❸ >>> pdfWatermarkReader = PyPDF2.PdfFileReader(open('watermark.pdf', 'rb'))
❹ >>> minutesFirstPage.mergePage(pdfWatermarkReader.getPage(0))
❺ >>> pdfWriter = PyPDF2.PdfFileWriter()
❻ >>> pdfWriter.addPage(minutesFirstPage)

❼ >>> for pageNum in range(1, pdfReader.numPages):
    pageObj = pdfReader.getPage(pageNum)
    pdfWriter.addPage(pageObj)
```

```
>>> resultPdfFile = open('watermarkedCover.pdf', 'wb')
>>> pdfWriter.write(resultPdfFile)
>>> minutesFile.close()
>>> resultPdfFile.close()
```

Here we make a PdfFileReader object of *meetingminutes.pdf* ❶. We call `getPage(0)` to get a Page object for the first page and store this object in `minutesFirstPage` ❷. We then make a PdfFileReader object for *watermark.pdf* ❸ and call `mergePage()` on `minutesFirstPage` ❹. The argument we pass to `mergePage()` is a Page object for the first page of *watermark.pdf*.

Now that we've called `mergePage()` on `minutesFirstPage`, `minutesFirstPage` represents the watermarked first page. We make a PdfFileWriter object ❺ and add the watermarked first page ❻. Then we loop through the rest of the pages in *meetingminutes.pdf* and add them to the PdfFileWriter object ❼. Finally, we open a new PDF called *watermarkedCover.pdf* and write the contents of the PdfFileWriter to the new PDF.

Figure 13-3 shows the results. Our new PDF, *watermarkedCover.pdf*, has all the contents of the *meetingminutes.pdf*, and the first page is watermarked.



Figure 13-3: The original PDF (left), the watermark PDF (center), and the merged PDF (right)

Encrypting PDFs

A PdfFileWriter object can also add encryption to a PDF document. Enter the following into the interactive shell:

```
>>> import PyPDF2
>>> pdfFile = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFile)
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> for pageNum in range(pdfReader.numPages):
    pdfWriter.addPage(pdfReader.getPage(pageNum))

❶ >>> pdfWriter.encrypt('swordfish')
>>> resultPdf = open('encryptedminutes.pdf', 'wb')
>>> pdfWriter.write(resultPdf)
>>> resultPdf.close()
```

Before calling the `write()` method to save to a file, call the `encrypt()` method and pass it a password string ❶. PDFs can have a *user password* (allowing you to view the PDF) and an *owner password* (allowing you to set permissions for printing, commenting, extracting text, and other features). The user password and owner password are the first and second arguments to `encrypt()`, respectively. If only one string argument is passed to `encrypt()`, it will be used for both passwords.

In this example, we copied the pages of *meetingminutes.pdf* to a `PdfFileWriter` object. We encrypted the `PdfFileWriter` with the password *swordfish*, opened a new PDF called *encryptedminutes.pdf*, and wrote the contents of the `PdfFileWriter` to the new PDF. Before anyone can view *encryptedminutes.pdf*, they'll have to enter this password. You may want to delete the original, unencrypted *meetingminutes.pdf* file after ensuring its copy was correctly encrypted.

Project: Combining Select Pages from Many PDFs

Say you have the boring job of merging several dozen PDF documents into a single PDF file. Each of them has a cover sheet as the first page, but you don't want the cover sheet repeated in the final result. Even though there are lots of free programs for combining PDFs, many of them simply merge entire files together. Let's write a Python program to customize which pages you want in the combined PDF.

At a high level, here's what the program will do:

- Find all PDF files in the current working directory.
- Sort the filenames so the PDFs are added in order.
- Write each page, excluding the first page, of each PDF to the output file.

In terms of implementation, your code will need to do the following:

- Call `os.listdir()` to find all the files in the working directory and remove any non-PDF files.
- Call Python's `sort()` list method to alphabetize the filenames.
- Create a `PdfFileWriter` object for the output PDF.
- Loop over each PDF file, creating a `PdfFileReader` object for it.
- Loop over each page (except the first) in each PDF file.
- Add the pages to the output PDF.
- Write the output PDF to a file named *allminutes.pdf*.

For this project, open a new file editor window and save it as *combinePdfs.py*.

Step 1: Find All PDF Files

First, your program needs to get a list of all files with the `.pdf` extension in the current working directory and sort them. Make your code look like the following:

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory into
# into a single PDF.

❶ import PyPDF2, os

# Get all the PDF filenames.
pdfFiles = []
for filename in os.listdir('.'):
    if filename.endswith('.pdf'):
❷        pdfFiles.append(filename)
❸ pdfFiles.sort(key=str.lower)

❹ pdfWriter = PyPDF2.PdfFileWriter()

# TODO: Loop through all the PDF files.

# TODO: Loop through all the pages (except the first) and add them.

# TODO: Save the resulting PDF to a file.
```

After the shebang line and the descriptive comment about what the program does, this code imports the `os` and `PyPDF2` modules ❶. The `os.listdir('.')` call will return a list of every file in the current working directory. The code loops over this list and adds only those files with the `.pdf` extension to `pdfFiles` ❷. Afterward, this list is sorted in alphabetical order with the `key=str.lower` keyword argument to `sort()` ❸.

A `PdfFileWriter` object is created to hold the combined PDF pages ❹. Finally, a few comments outline the rest of the program.

Step 2: Open Each PDF

Now the program must read each PDF file in `pdfFiles`. Add the following to your program:

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory into
# into a single PDF.

import PyPDF2, os

# Get all the PDF filenames.
pdfFiles = []
--snip--
```

```
# Loop through all the PDF files.
for filename in pdfFiles:
    pdfFileObj = open(filename, 'rb')
    pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
    # TODO: Loop through all the pages (except the first) and add them.

# TODO: Save the resulting PDF to a file.
```

For each PDF, the loop opens a filename in read-binary mode by calling `open()` with `'rb'` as the second argument. The `open()` call returns a `File` object, which gets passed to `PyPDF2.PdfFileReader()` to create a `PdfFileReader` object for that PDF file.

Step 3: Add Each Page

For each PDF, you'll want to loop over every page except the first. Add this code to your program:

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory into
# a single PDF.

import PyPDF2, os

--snip--

# Loop through all the PDF files.
for filename in pdfFiles:
--snip--
    # Loop through all the pages (except the first) and add them.
❶    for pageNum in range(1, pdfReader.numPages):
        pageObj = pdfReader.getPage(pageNum)
        pdfWriter.addPage(pageObj)

# TODO: Save the resulting PDF to a file.
```

The code inside the `for` loop copies each `Page` object individually to the `PdfFileWriter` object. Remember, you want to skip the first page. Since `PyPDF2` considers 0 to be the first page, your loop should start at 1 ❶ and then go up to, but not include, the integer in `pdfReader.numPages`.

Step 4: Save the Results

After these nested `for` loops are done looping, the `pdfWriter` variable will contain a `PdfFileWriter` object with the pages for all the PDFs combined. The last step is to write this content to a file on the hard drive. Add this code to your program:

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory into
# a single PDF.
```

```
import PyPDF2, os

--snip--

# Loop through all the PDF files.
for filename in pdfFiles:
    --snip--
        # Loop through all the pages (except the first) and add them.
        for pageNum in range(1, pdfReader.numPages):
            --snip--

# Save the resulting PDF to a file.
pdfOutput = open('allminutes.pdf', 'wb')
pdfWriter.write(pdfOutput)
pdfOutput.close()
```

Passing 'wb' to `open()` opens the output PDF file, `allminutes.pdf`, in write-binary mode. Then, passing the resulting `File` object to the `write()` method creates the actual PDF file. A call to the `close()` method finishes the program.

Ideas for Similar Programs

Being able to create PDFs from the pages of other PDFs will let you make programs that can do the following:

- Cut out specific pages from PDFs.
- Reorder pages in a PDF.
- Create a PDF from only those pages that have some specific text, identified by `extractText()`.

Word Documents

Python can create and modify Word documents, which have the `.docx` file extension, with the `python-docx` module. You can install the module by running `pip install python-docx`. (Appendix A has full details on installing third-party modules.)

NOTE

When using pip to first install Python-Docx, be sure to install python-docx, not docx. The installation name docx is for a different module that this book does not cover. However, when you are going to import the python-docx module, you'll need to run import docx, not import python-docx.

If you don't have Word, LibreOffice Writer and OpenOffice Writer are both free alternative applications for Windows, OS X, and Linux that can be used to open `.docx` files. You can download them from <https://www.libreoffice.org> and <http://openoffice.org>, respectively. The full documentation for Python-Docx is available at <https://python-docx.readthedocs.org/>. Although there is a version of Word for OS X, this chapter will focus on Word for Windows.

Compared to plaintext, *.docx* files have a lot of structure. This structure is represented by three different data types in Python-Docx. At the highest level, a `Document` object represents the entire document. The `Document` object contains a list of `Paragraph` objects for the paragraphs in the document. (A new paragraph begins whenever the user presses ENTER or RETURN while typing in a Word document.) Each of these `Paragraph` objects contains a list of one or more `Run` objects. The single-sentence paragraph in Figure 13-4 has four runs.



Figure 13-4: The `Run` objects identified in a `Paragraph` object

The text in a Word document is more than just a string. It has font, size, color, and other styling information associated with it. A *style* in Word is a collection of these attributes. A `Run` object is a contiguous run of text with the same style. A new `Run` object is needed whenever the text style changes.

Reading Word Documents

Let's experiment with the `python-docx` module. Download `demo.docx` from <http://nostarch.com/automatestuff/> and save the document to the working directory. Then enter the following into the interactive shell:

```
>>> import docx
❶ >>> doc = docx.Document('demo.docx')
❷ >>> len(doc.paragraphs)
7
❸ >>> doc.paragraphs[0].text
'Document Title'
❹ >>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
❺ >>> len(doc.paragraphs[1].runs)
4
❻ >>> doc.paragraphs[1].runs[0].text
'A plain paragraph with some '
❼ >>> doc.paragraphs[1].runs[1].text
'bold'
❽ >>> doc.paragraphs[1].runs[2].text
' and some '
❾ >>> doc.paragraphs[1].runs[3].text
'italic'
```

At ❶, we open a *.docx* file in Python, call `docx.Document()`, and pass the filename `demo.docx`. This will return a `Document` object, which has a `paragraphs` attribute that is a list of `Paragraph` objects. When we call `len()` on `doc.paragraphs`, it returns 7, which tells us that there are seven `Paragraph` objects in this document ❷. Each of these `Paragraph` objects has a `text`

attribute that contains a string of the text in that paragraph (without the style information). Here, the first text attribute contains 'DocumentTitle' ❸, and the second contains 'A plain paragraph with some bold and some italic' ❹.

Each Paragraph object also has a runs attribute that is a list of Run objects. Run objects also have a text attribute, containing just the text in that particular run. Let's look at the text attributes in the second Paragraph object, 'A plain paragraph with some bold and some italic'. Calling len() on this Paragraph object tells us that there are four Run objects ❺. The first run object contains 'A plain paragraph with some ' ❻. Then, the text changes to a bold style, so 'bold' starts a new Run object ❼. The text returns to an unbolded style after that, which results in a third Run object, ' and some ' ❽. Finally, the fourth and last Run object contains 'italic' in an italic style ❾.

With Python-Docx, your Python programs will now be able to read the text from a .docx file and use it just like any other string value.

Getting the Full Text from a .docx File

If you care only about the text, not the styling information, in the Word document, you can use the getText() function. It accepts a filename of a .docx file and returns a single string value of its text. Open a new file editor window and enter the following code, saving it as *readDocx.py*:

```
#! python3

import docx

def getText(filename):
    doc = docx.Document(filename)
    fullText = []
    for para in doc.paragraphs:
        fullText.append(para.text)
    return '\n'.join(fullText)
```

The getText() function opens the Word document, loops over all the Paragraph objects in the paragraphs list, and then appends their text to the list in fullText. After the loop, the strings in fullText are joined together with newline characters.

The *readDocx.py* program can be imported like any other module. Now if you just need the text from a Word document, you can enter the following:

```
>>> import readDocx
>>> print(readDocx.getText('demo.docx'))
Document Title
A plain paragraph with some bold and some italic
Heading, level 1
Intense quote
first item in unordered list
first item in ordered list
```

You can also adjust `getText()` to modify the string before returning it. For example, to indent each paragraph, replace the `append()` call in `readDocx.py` with this:

```
fullText.append(' ' + para.text)
```

To add a double space in between paragraphs, change the `join()` call code to this:

```
return '\n\n'.join(fullText)
```

As you can see, it takes only a few lines of code to write functions that will read a `.docx` file and return a string of its content to your liking.

Styling Paragraph and Run Objects

In Word for Windows, you can see the styles by pressing **CTRL-ALT-SHIFT-S** to display the Styles pane, which looks like Figure 13-5. On OS X, you can view the Styles pane by clicking the **View ▶ Styles** menu item.



Figure 13-5: Display the Styles pane by pressing **CTRL-ALT-SHIFT-S** on Windows.

Word and other word processors use styles to keep the visual presentation of similar types of text consistent and easy to change. For example, perhaps you want to set body paragraphs in 11-point, Times New Roman, left-justified, ragged-right text. You can create a style with these settings and

assign it to all body paragraphs. Then, if you later want to change the presentation of all body paragraphs in the document, you can just change the style, and all those paragraphs will be automatically updated.

For Word documents, there are three types of styles: *Paragraph styles* can be applied to Paragraph objects, *character styles* can be applied to Run objects, and *linked styles* can be applied to both kinds of objects. You can give both Paragraph and Run objects styles by setting their style attribute to a string. This string should be the name of a style. If style is set to None, then there will be no style associated with the Paragraph or Run object.

The string values for the default Word styles are as follows:

'Normal'	'Heading5'	'ListBullet'	'ListParagraph'
'BodyText'	'Heading6'	'ListBullet2'	'MacroText'
'BodyText2'	'Heading7'	'ListBullet3'	'NoSpacing'
'BodyText3'	'Heading8'	'ListContinue'	'Quote'
'Caption'	'Heading9'	'ListContinue2'	'Subtitle'
'Heading1'	'IntenseQuote'	'ListContinue3'	'TOCHeading'
'Heading2'	'List'	'ListNumber'	'Title'
'Heading3'	'List2'	'ListNumber2'	
'Heading4'	'List3'	'ListNumber3'	

When setting the style attribute, do not use spaces in the style name. For example, while the style name may be Subtle Emphasis, you should set the style attribute to the string value 'SubtleEmphasis' instead of 'Subtle Emphasis'. Including spaces will cause Word to misread the style name and not apply it.

When using a linked style for a Run object, you will need to add 'Char' to the end of its name. For example, to set the Quote linked style for a Paragraph object, you would use `paragraphObj.style = 'Quote'`, but for a Run object, you would use `runObj.style = 'QuoteChar'`.

In the current version of Python-Docx (0.7.4), the only styles that can be used are the default Word styles and the styles in the opened `.docx`. New styles cannot be created—though this may change in future versions of Python-Docx.

Creating Word Documents with Nondefault Styles

If you want to create Word documents that use styles beyond the default ones, you will need to open Word to a blank Word document and create the styles yourself by clicking the **New Style** button at the bottom of the Styles pane (Figure 13-6 shows this on Windows).

This will open the Create New Style from Formatting dialog, where you can enter the new style. Then, go back into the interactive shell and open this blank Word document with `docx.Document()`, using it as the base for your Word document. The name you gave this style will now be available to use with Python-Docx.



Figure 13-6: The New Style button (left) and the Create New Style from Formatting dialog (right)

Run Attributes

Runs can be further styled using text attributes. Each attribute can be set to one of three values: True (the attribute is always enabled, no matter what other styles are applied to the run), False (the attribute is always disabled), or None (defaults to whatever the run's style is set to).

Table 13-1 lists the text attributes that can be set on Run objects.

Table 13-1: Run Object text Attributes

Attribute	Description
bold	The text appears in bold.
italic	The text appears in italic.
underline	The text is underlined.
strike	The text appears with strikethrough.
double_strike	The text appears with double strikethrough.
all_caps	The text appears in capital letters.
small_caps	The text appears in capital letters, with lowercase letters two points smaller.
shadow	The text appears with a shadow.
outline	The text appears outlined rather than solid.
rtl	The text is written right-to-left.
imprint	The text appears pressed into the page.
emboss	The text appears raised off the page in relief.

For example, to change the styles of *demo.docx*, enter the following into the interactive shell:

```
>>> doc = docx.Document('demo.docx')
>>> doc.paragraphs[0].text
'Document Title'
>>> doc.paragraphs[0].style
'Title'
>>> doc.paragraphs[0].style = 'Normal'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
>>> (doc.paragraphs[1].runs[0].text, doc.paragraphs[1].runs[1].text, doc.
paragraphs[1].runs[2].text, doc.paragraphs[1].runs[3].text)
('A plain paragraph with some ', 'bold', ' and some ', 'italic')
>>> doc.paragraphs[1].runs[0].style = 'QuoteChar'
>>> doc.paragraphs[1].runs[1].underline = True
>>> doc.paragraphs[1].runs[3].underline = True
>>> doc.save('restyled.docx')
```

Here, we use the `text` and `style` attributes to easily see what's in the paragraphs in our document. We can see that it's simple to divide a paragraph into runs and access each run individually. So we get the first, second, and fourth runs in the second paragraph, style each run, and save the results to a new document.

The words *Document Title* at the top of *restyled.docx* will have the Normal style instead of the Title style, the Run object for the text *A plain paragraph with some* will have the QuoteChar style, and the two Run objects for the words *bold* and *italic* will have their `underline` attributes set to `True`. Figure 13-7 shows how the styles of paragraphs and runs look in *restyled.docx*.



Figure 13-7: The *restyled.docx* file

You can find more complete documentation on Python-Docx's use of styles at <https://python-docx.readthedocs.org/en/latest/user/styles.html>.

Writing Word Documents

Enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
<docx.text.Paragraph object at 0x0000000003B56F60>
>>> doc.save('helloworld.docx')
```

To create your own `.docx` file, call `docx.Document()` to return a new, blank Word Document object. The `add_paragraph()` document method adds a new paragraph of text to the document and returns a reference to the Paragraph object that was added. When you're done adding text, pass a filename string to the `save()` document method to save the Document object to a file.

This will create a file named `helloworld.docx` in the current working directory that, when opened, looks like Figure 13-8.



Figure 13-8: The Word document created using `add_paragraph('Hello world!')`

You can add paragraphs by calling the `add_paragraph()` method again with the new paragraph's text. Or to add text to the end of an existing paragraph, you can call the paragraph's `add_run()` method and pass it a string. Enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
<docx.text.Paragraph object at 0x000000000366AD30>
>>> paraObj1 = doc.add_paragraph('This is a second paragraph.')
>>> paraObj2 = doc.add_paragraph('This is a yet another paragraph.')
>>> paraObj1.add_run(' This text is being added to the second paragraph.')
<docx.text.Run object at 0x0000000003A2C860>
>>> doc.save('multipleParagraphs.docx')
```

The resulting document will look like Figure 13-9. Note that the text *This text is being added to the second paragraph.* was added to the Paragraph object in `paraObj1`, which was the second paragraph added to `doc`. The `add_paragraph()` and `add_run()` functions return Paragraph and Run objects, respectively, to save you the trouble of extracting them as a separate step.

Keep in mind that as of Python-Docx version 0.5.3, new Paragraph objects can be added only to the end of the document, and new Run objects can be added only to the end of a Paragraph object.

The `save()` method can be called again to save the additional changes you've made.



Figure 13-9: The document with multiple Paragraph and Run objects added

Both `add_paragraph()` and `add_run()` accept an optional second argument that is a string of the Paragraph or Run object's style. For example:

```
>>> doc.add_paragraph('Hello world!', 'Title')
```

This line adds a paragraph with the text *Hello world!* in the Title style.

Adding Headings

Calling `add_heading()` adds a paragraph with one of the heading styles. Enter the following into the interactive shell:

```
>>> doc = docx.Document()
>>> doc.add_heading('Header 0', 0)
<docx.text.Paragraph object at 0x000000000036CB3C8>
>>> doc.add_heading('Header 1', 1)
<docx.text.Paragraph object at 0x000000000036CB630>
>>> doc.add_heading('Header 2', 2)
<docx.text.Paragraph object at 0x000000000036CB828>
>>> doc.add_heading('Header 3', 3)
<docx.text.Paragraph object at 0x000000000036CB2E8>
>>> doc.add_heading('Header 4', 4)
<docx.text.Paragraph object at 0x000000000036CB3C8>
>>> doc.save('headings.docx')
```

The arguments to `add_heading()` are a string of the heading text and an integer from 0 to 4. The integer 0 makes the heading the Title style, which is used for the top of the document. Integers 1 to 4 are for various heading levels, with 1 being the main heading and 4 the lowest subheading. The `add_heading()` function returns a Paragraph object to save you the step of extracting it from the Document object as a separate step.

The resulting *headings.docx* file will look like Figure 13-10.

Header 0

Header 1

Header 2

Header 3

Header 4

Figure 13-10: The *headings.docx* document with headings 0 to 4

Adding Line and Page Breaks

To add a line break (rather than starting a whole new paragraph), you can call the `add_break()` method on the `Run` object you want to have the break appear after. If you want to add a page break instead, you need to pass the value `docx.text.WD_BREAK.PAGE` as a lone argument to `add_break()`, as is done in the middle of the following example:

```
>>> doc = docx.Document()
>>> doc.add_paragraph('This is on the first page!')
<docx.text.Paragraph object at 0x0000000003785518>
❶ >>> doc.paragraphs[0].runs[0].add_break(docx.text.WD_BREAK.PAGE)
>>> doc.add_paragraph('This is on the second page!')
<docx.text.Paragraph object at 0x00000000037855F8>
>>> doc.save('twoPage.docx')
```

This creates a two-page Word document with *This is on the first page!* on the first page and *This is on the second page!* on the second. Even though there was still plenty of space on the first page after the text *This is on the first page!*, we forced the next paragraph to begin on a new page by inserting a page break after the first run of the first paragraph ❶.

Adding Pictures

Document objects have an `add_picture()` method that will let you add an image to the end of the document. Say you have a file *zophie.png* in the current working directory. You can add *zophie.png* to the end of your document with a width of 1 inch and height of 4 centimeters (Word can use both imperial and metric units) by entering the following:

```
>>> doc.add_picture('zophie.png', width=docx.shared.Inches(1),
height=docx.shared.Cm(4))
<docx.shape.InlineShape object at 0x00000000036C7D30>
```

The first argument is a string of the image’s filename. The optional `width` and `height` keyword arguments will set the width and height of the image in the document. If left out, the width and height will default to the normal size of the image.

You’ll probably prefer to specify an image’s height and width in familiar units such as inches and centimeters, so you can use the `docx.shared.Inches()` and `docx.shared.Cm()` functions when you’re specifying the `width` and `height` keyword arguments.

Summary

Text information isn’t just for plaintext files; in fact, it’s pretty likely that you deal with PDFs and Word documents much more often. You can use the PyPDF2 module to read and write PDF documents. Unfortunately, reading text from PDF documents might not always result in a perfect translation to a string because of the complicated PDF file format, and some PDFs might not be readable at all. In these cases, you’re out of luck unless future updates to PyPDF2 support additional PDF features.

Word documents are more reliable, and you can read them with the `python-docx` module. You can manipulate text in Word documents via `Paragraph` and `Run` objects. These objects can also be given styles, though they must be from the default set of styles or styles already in the document. You can add new paragraphs, headings, breaks, and pictures to the document, though only to the end.

Many of the limitations that come with working with PDFs and Word documents are because these formats are meant to be nicely displayed for human readers, rather than easy to parse by software. The next chapter takes a look at two other common formats for storing information: JSON and CSV files. These formats are designed to be used by computers, and you’ll see that Python can work with these formats much more easily.

Practice Questions

1. A string value of the PDF filename is *not* passed to the `PyPDF2.PdfFileReader()` function. What do you pass to the function instead?
2. What modes do the `File` objects for `PdfFileReader()` and `PdfFileWriter()` need to be opened in?
3. How do you acquire a `Page` object for page 5 from a `PdfFileReader` object?
4. What `PdfFileReader` variable stores the number of pages in the PDF document?
5. If a `PdfFileReader` object’s PDF is encrypted with the password `swordfish`, what must you do before you can obtain `Page` objects from it?
6. What methods do you use to rotate a page?
7. What method returns a `Document` object for a file named `demo.docx`?

8. What is the difference between a Paragraph object and a Run object?
9. How do you obtain a list of Paragraph objects for a Document object that's stored in a variable named doc?
10. What type of object has bold, underline, italic, strike, and outline variables?
11. What is the difference between setting the bold variable to True, False, or None?
12. How do you create a Document object for a new Word document?
13. How do you add a paragraph with the text 'Hello there!' to a Document object stored in a variable named doc?
14. What integers represent the levels of headings available in Word documents?

Practice Projects

For practice, write programs that do the following.

PDF Paranoia

Using the `os.walk()` function from Chapter 9, write a script that will go through every PDF in a folder (and its subfolders) and encrypt the PDFs using a password provided on the command line. Save each encrypted PDF with an `_encrypted.pdf` suffix added to the original filename. Before deleting the original file, have the program attempt to read and decrypt the file to ensure that it was encrypted correctly.

Then, write a program that finds all encrypted PDFs in a folder (and its subfolders) and creates a decrypted copy of the PDF using a provided password. If the password is incorrect, the program should print a message to the user and continue to the next PDF.

Custom Invitations as Word Documents

Say you have a text file of guest names. This `guests.txt` file has one name per line, as follows:

Prof. Plum
Miss Scarlet
Col. Mustard
Al Sweigart
RoboCop

Write a program that would generate a Word document with custom invitations that look like Figure 13-11.

Since Python-Docx can use only those styles that already exist in the Word document, you will have to first add these styles to a blank Word file and then open that file with Python-Docx. There should be one invitation per page in the resulting Word document, so call `add_break()` to add a page break after the last paragraph of each invitation. This way, you will need to open only one Word document to print all of the invitations at once.



Figure 13-11: The Word document generated by your custom invite script

You can download a sample *guests.txt* file from <http://nostarch.com/automatestuff/>.

Brute-Force PDF Password Breaker

Say you have an encrypted PDF that you have forgotten the password to, but you remember it was a single English word. Trying to guess your forgotten password is quite a boring task. Instead you can write a program that will decrypt the PDF by trying every possible English word until it finds one that works. This is called a *brute-force password attack*. Download the text file *dictionary.txt* from <http://nostarch.com/automatestuff/>. This *dictionary file* contains over 44,000 English words with one word per line.

Using the file-reading skills you learned in Chapter 8, create a list of word strings by reading this file. Then loop over each word in this list, passing it to the *decrypt()* method. If this method returns the integer 0, the password was wrong and your program should continue to the next password. If *decrypt()* returns 1, then your program should break out of the loop and print the hacked password. You should try both the uppercase and lowercase form of each word. (On my laptop, going through all 88,000 uppercase and lowercase words from the dictionary file takes a couple of minutes. This is why you shouldn't use a simple English word for your passwords.)

14

WORKING WITH CSV FILES AND JSON DATA



In Chapter 13, you learned how to extract text from PDF and Word documents. These files were in a binary format, which required special Python modules to access their data.

CSV and JSON files, on the other hand, are just plain-text files. You can view them in a text editor, such as IDLE’s file editor. But Python also comes with the special `csv` and `json` modules, each providing functions to help you work with these file formats.

CSV stands for “comma-separated values,” and CSV files are simplified spreadsheets stored as plaintext files. Python’s `csv` module makes it easy to parse CSV files.

JSON (pronounced “JAY-sawn” or “Jason”—it doesn’t matter how because either way people will say you’re pronouncing it wrong) is a format that stores information as JavaScript source code in plaintext files.

(JSON is short for JavaScript Object Notation.) You don’t need to know the JavaScript programming language to use JSON files, but the JSON format is useful to know because it’s used in many web applications.

The `csv` Module

Each line in a CSV file represents a row in the spreadsheet, and commas separate the cells in the row. For example, the spreadsheet *example.xlsx* from <http://nostarch.com/automatestuff/> would look like this in a CSV file:

```
4/5/2015 13:34,Apples,73
4/5/2015 3:41,Cherries,85
4/6/2015 12:46,Pears,14
4/8/2015 8:59,Oranges,52
4/10/2015 2:07,Apples,152
4/10/2015 18:10,Bananas,23
4/10/2015 2:40,Strawberries,98
```

I will use this file for this chapter’s interactive shell examples. You can download *example.csv* from <http://nostarch.com/automatestuff/> or enter the text into a text editor and save it as *example.csv*.

CSV files are simple, lacking many of the features of an Excel spreadsheet. For example, CSV files

- Don’t have types for their values—everything is a string
- Don’t have settings for font size or color
- Don’t have multiple worksheets
- Can’t specify cell widths and heights
- Can’t have merged cells
- Can’t have images or charts embedded in them

The advantage of CSV files is simplicity. CSV files are widely supported by many types of programs, can be viewed in text editors (including IDLE’s file editor), and are a straightforward way to represent spreadsheet data. The CSV format is exactly as advertised: It’s just a text file of comma-separated values.

Since CSV files are just text files, you might be tempted to read them in as a string and then process that string using the techniques you learned in Chapter 8. For example, since each cell in a CSV file is separated by a comma, maybe you could just call the `split()` method on each line of text to get the values. But not every comma in a CSV file represents the boundary between two cells. CSV files also have their own set of escape characters to allow commas and other characters to be included *as part of the values*. The `split()` method doesn’t handle these escape characters. Because of these potential pitfalls, you should always use the `csv` module for reading and writing CSV files.

Reader Objects

To read data from a CSV file with the `csv` module, you need to create a Reader object. A Reader object lets you iterate over lines in the CSV file. Enter the following into the interactive shell, with `example.csv` in the current working directory:

```
❶ >>> import csv
❷ >>> exampleFile = open('example.csv')
❸ >>> exampleReader = csv.reader(exampleFile)
❹ >>> exampleData = list(exampleReader)
❺ >>> exampleData
[['4/5/2015 13:34', 'Apples', '73'], ['4/5/2015 3:41', 'Cherries', '85'],
['4/6/2015 12:46', 'Pears', '14'], ['4/8/2015 8:59', 'Oranges', '52'],
['4/10/2015 2:07', 'Apples', '152'], ['4/10/2015 18:10', 'Bananas', '23'],
['4/10/2015 2:40', 'Strawberries', '98']]
```

The `csv` module comes with Python, so we can import it ❶ without having to install it first.

To read a CSV file with the `csv` module, first open it using the `open()` function ❷, just as you would any other text file. But instead of calling the `read()` or `readlines()` method on the `File` object that `open()` returns, pass it to the `csv.reader()` function ❸. This will return a Reader object for you to use. Note that you don't pass a filename string directly to the `csv.reader()` function.

The most direct way to access the values in the Reader object is to convert it to a plain Python list by passing it to `list()` ❹. Using `list()` on this Reader object returns a list of lists, which you can store in a variable like `exampleData`. Entering `exampleData` in the shell displays the list of lists ❺.

Now that you have the CSV file as a list of lists, you can access the value at a particular row and column with the expression `exampleData[row][col]`, where `row` is the index of one of the lists in `exampleData`, and `col` is the index of the item you want from that list. Enter the following into the interactive shell:

```
>>> exampleData[0][0]
'4/5/2015 13:34'
>>> exampleData[0][1]
'Apples'
>>> exampleData[0][2]
'73'
>>> exampleData[1][1]
'Cherries'
>>> exampleData[6][1]
'Strawberries'
```

`exampleData[0][0]` goes into the first list and gives us the first string, `exampleData[0][2]` goes into the first list and gives us the third string, and so on.

Reading Data from Reader Objects in a for Loop

For large CSV files, you'll want to use the Reader object in a for loop. This avoids loading the entire file into memory at once. For example, enter the following into the interactive shell:

```
>>> import csv
>>> exampleFile = open('example.csv')
>>> exampleReader = csv.reader(exampleFile)
>>> for row in exampleReader:
    print('Row #' + str(exampleReader.line_num) + ' ' + str(row))

Row #1 ['4/5/2015 13:34', 'Apples', '73']
Row #2 ['4/5/2015 3:41', 'Cherries', '85']
Row #3 ['4/6/2015 12:46', 'Pears', '14']
Row #4 ['4/8/2015 8:59', 'Oranges', '52']
Row #5 ['4/10/2015 2:07', 'Apples', '152']
Row #6 ['4/10/2015 18:10', 'Bananas', '23']
Row #7 ['4/10/2015 2:40', 'Strawberries', '98']
```

After you import the csv module and make a Reader object from the CSV file, you can loop through the rows in the Reader object. Each row is a list of values, with each value representing a cell.

The print() function call prints the number of the current row and the contents of the row. To get the row number, use the Reader object's line_num variable, which contains the number of the current line.

The Reader object can be looped over only once. To reread the CSV file, you must call csv.reader to create a Reader object.

Writer Objects

A Writer object lets you write data to a CSV file. To create a Writer object, you use the csv.writer() function. Enter the following into the interactive shell:

```
>>> import csv
❶ >>> outputFile = open('output.csv', 'w', newline='')
❷ >>> outputWriter = csv.writer(outputFile)
>>> outputWriter.writerow(['spam', 'eggs', 'bacon', 'ham'])
21
>>> outputWriter.writerow(['Hello, world!', 'eggs', 'bacon', 'ham'])
32
>>> outputWriter.writerow([1, 2, 3.141592, 4])
16
>>> outputFile.close()
```

First, call open() and pass it 'w' to open a file in write mode ❶. This will create the object you can then pass to csv.writer() ❷ to create a Writer object.

On Windows, you'll also need to pass a blank string for the open() function's newline keyword argument. For technical reasons beyond the scope of this book, if you forget to set the newline argument, the rows in *output.csv* will be double-spaced, as shown in Figure 14-1.

The screenshot shows a Microsoft Excel spreadsheet with the following data:

	A	B	C	D	E	F	G
1	42	2	3	4	5	6	7
2							
3	2	4	6	8	10	12	14
4							
5	3	6	9	12	15	18	21
6							
7	4	8	12	16	20	24	28
8							
9	5	10	15	20	25	30	35
10							

Figure 14-1: If you forget the `newline=''` keyword argument in `open()`, the CSV file will be double-spaced.

The `writerow()` method for `Writer` objects takes a list argument. Each value in the list is placed in its own cell in the output CSV file. The return value of `writerow()` is the number of characters written to the file for that row (including newline characters).

This code produces an `output.csv` file that looks like this:

```
spam,eggs,bacon,ham
"Hello, world!",eggs,bacon,ham
1,2,3.141592,4
```

Notice how the `Writer` object automatically escapes the comma in the value 'Hello, world!' with double quotes in the CSV file. The `csv` module saves you from having to handle these special cases yourself.

The delimiter and lineterminator Keyword Arguments

Say you want to separate cells with a tab character instead of a comma and you want the rows to be double-spaced. You could enter something like the following into the interactive shell:

```
>>> import csv
>>> csvFile = open('example.tsv', 'w', newline='')
❶ >>> csvWriter = csv.writer(csvFile, delimiter='\t', lineterminator='\n\n')
>>> csvWriter.writerow(['apples', 'oranges', 'grapes'])
24
>>> csvWriter.writerow(['eggs', 'bacon', 'ham'])
17
>>> csvWriter.writerow(['spam', 'spam', 'spam', 'spam', 'spam', 'spam'])
32
>>> csvFile.close()
```

This changes the delimiter and line terminator characters in your file. The *delimiter* is the character that appears between cells on a row. By default, the delimiter for a CSV file is a comma. The *line terminator* is the character that comes at the end of a row. By default, the line terminator is a newline. You can change characters to different values by using the `delimiter` and `lineterminator` keyword arguments with `csv.writer()`.

Passing `delimiter='\\t'` and `lineterminator='\\n\\n'` ❶ changes the character between cells to a tab and the character between rows to two newlines. We then call `writerow()` three times to give us three rows.

This produces a file named `example.tsv` with the following contents:

apples	oranges	grapes
eggs	bacon	ham
spam	spam	spam
spam	spam	spam

Now that our cells are separated by tabs, we're using the file extension `.tsv`, for tab-separated values.

Project: Removing the Header from CSV Files

Say you have the boring job of removing the first line from several hundred CSV files. Maybe you'll be feeding them into an automated process that requires just the data and not the headers at the top of the columns. You *could* open each file in Excel, delete the first row, and resave the file—but that would take hours. Let's write a program to do it instead.

The program will need to open every file with the `.csv` extension in the current working directory, read in the contents of the CSV file, and rewrite the contents without the first row to a file of the same name. This will replace the old contents of the CSV file with the new, headless contents.

WARNING

As always, whenever you write a program that modifies files, be sure to back up the files, first just in case your program does not work the way you expect it to. You don't want to accidentally erase your original files.

At a high level, the program must do the following:

- Find all the CSV files in the current working directory.
- Read in the full contents of each file.
- Write out the contents, skipping the first line, to a new CSV file.

At the code level, this means the program will need to do the following:

- Loop over a list of files from `os.listdir()`, skipping the non-CSV files.
- Create a CSV Reader object and read in the contents of the file, using the `line_num` attribute to figure out which line to skip.
- Create a CSV Writer object and write out the read-in data to the new file.

For this project, open a new file editor window and save it as `removeCsvHeader.py`.

Step 1: Loop Through Each CSV File

The first thing your program needs to do is loop over a list of all CSV filenames for the current working directory. Make your `removeCsvHeader.py` look like this:

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.

import csv, os

os.makedirs('headerRemoved', exist_ok=True)

# Loop through every file in the current working directory.
for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
        continue      # skip non-csv files
❶    print('Removing header from ' + csvFilename + '...')

    # TODO: Read the CSV file in (skipping first row).

    # TODO: Write out the CSV file.
```

The `os.makedirs()` call will create a `headerRemoved` folder where all the headerless CSV files will be written. A `for` loop on `os.listdir('.')` gets you partway there, but it will loop over *all* files in the working directory, so you'll need to add some code at the start of the loop that skips filenames that don't end with `.csv`. The `continue` statement ❶ makes the `for` loop move on to the next filename when it comes across a non-CSV file.

Just so there's *some* output as the program runs, print out a message saying which CSV file the program is working on. Then, add some `TODO` comments for what the rest of the program should do.

Step 2: Read in the CSV File

The program doesn't remove the first line from the CSV file. Rather, it creates a new copy of the CSV file without the first line. Since the copy's filename is the same as the original filename, the copy will overwrite the original.

The program will need a way to track whether it is currently looping on the first row. Add the following to `removeCsvHeader.py`.

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.

--snip--
```

```
# Read the CSV file in (skipping first row).
csvRows = []
csvFileObj = open(csvFilename)
readerObj = csv.reader(csvFileObj)
for row in readerObj:
    if readerObj.line_num == 1:
        continue    # skip first row
    csvRows.append(row)
csvFileObj.close()

# TODO: Write out the CSV file.
```

The Reader object's `line_num` attribute can be used to determine which line in the CSV file it is currently reading. Another `for` loop will loop over the rows returned from the CSV Reader object, and all rows but the first will be appended to `csvRows`.

As the `for` loop iterates over each row, the code checks whether `readerObj.line_num` is set to 1. If so, it executes a `continue` to move on to the next row without appending it to `csvRows`. For every row afterward, the condition will be always be `False`, and the row will be appended to `csvRows`.

Step 3: Write Out the CSV File Without the First Row

Now that `csvRows` contains all rows but the first row, the list needs to be written out to a CSV file in the `headerRemoved` folder. Add the following to `removeCsvHeader.py`:

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.

--snip--

❶ for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
        continue    # skip non-CSV files

--snip--

# Write out the CSV file.
csvFileObj = open(os.path.join('headerRemoved', csvFilename), 'w',
                  newline='')
csvWriter = csv.writer(csvFileObj)
for row in csvRows:
    csvWriter.writerow(row)
csvFileObj.close()
```

The CSV Writer object will write the list to a CSV file in `headerRemoved` using `csvFilename` (which we also used in the CSV reader). This will overwrite the original file.

Once we create the `Writer` object, we loop over the sublists stored in `csvRows` and write each sublist to the file.

After the code is executed, the outer `for` loop ❶ will loop to the next filename from `os.listdir('.')`. When that loop is finished, the program will be complete.

To test your program, download `removeCsvHeader.zip` from <http://nostarch.com/automatestuff/> and unzip it to a folder. Run the `removeCsvHeader.py` program in that folder. The output will look like this:

```
Removing header from NAICS_data_1048.csv...
Removing header from NAICS_data_1218.csv...
--snip--
Removing header from NAICS_data_9834.csv...
Removing header from NAICS_data_9986.csv...
```

This program should print a filename each time it strips the first line from a CSV file.

Ideas for Similar Programs

The programs that you could write for CSV files are similar to the kinds you could write for Excel files, since they're both spreadsheet files. You could write programs to do the following:

- Compare data between different rows in a CSV file or between multiple CSV files.
- Copy specific data from a CSV file to an Excel file, or vice versa.
- Check for invalid data or formatting mistakes in CSV files and alert the user to these errors.
- Read data from a CSV file as input for your Python programs.

JSON and APIs

JavaScript Object Notation is a popular way to format data as a single human-readable string. JSON is the native way that JavaScript programs write their data structures and usually resembles what Python's `pprint()` function would produce. You don't need to know JavaScript in order to work with JSON-formatted data.

Here's an example of data formatted as JSON:

```
{"name": "Zophie", "isCat": true,
 "miceCaught": 0, "napsTaken": 37.5,
 "felineIQ": null}
```

JSON is useful to know, because many websites offer JSON content as a way for programs to interact with the website. This is known as providing an *application programming interface (API)*. Accessing an API is the same

as accessing any other web page via a URL. The difference is that the data returned by an API is formatted (with JSON, for example) for machines; APIs aren't easy for people to read.

Many websites make their data available in JSON format. Facebook, Twitter, Yahoo, Google, Tumblr, Wikipedia, Flickr, Data.gov, Reddit, IMDb, Rotten Tomatoes, LinkedIn, and many other popular sites offer APIs for programs to use. Some of these sites require registration, which is almost always free. You'll have to find documentation for what URLs your program needs to request in order to get the data you want, as well as the general format of the JSON data structures that are returned. This documentation should be provided by whatever site is offering the API; if they have a "Developers" page, look for the documentation there.

Using APIs, you could write programs that do the following:

- Scrape raw data from websites. (Accessing APIs is often more convenient than downloading web pages and parsing HTML with Beautiful Soup.)
- Automatically download new posts from one of your social network accounts and post them to another account. For example, you could take your Tumblr posts and post them to Facebook.
- Create a "movie encyclopedia" for your personal movie collection by pulling data from IMDb, Rotten Tomatoes, and Wikipedia and putting it into a single text file on your computer.

You can see some examples of JSON APIs in the resources at <http://nostarch.com/automatestuff/>.

The `json` Module

Python's `json` module handles all the details of translating between a string with JSON data and Python values for the `json.loads()` and `json.dumps()` functions. JSON can't store *every* kind of Python value. It can contain values of only the following data types: strings, integers, floats, Booleans, lists, dictionaries, and `NoneType`. JSON cannot represent Python-specific objects, such as `File` objects, `CSV Reader` or `Writer` objects, `Regex` objects, or `Selenium WebElement` objects.

Reading JSON with the `loads()` Function

To translate a string containing JSON data into a Python value, pass it to the `json.loads()` function. (The name means "load string," not "loads.") Enter the following into the interactive shell:

```
>>> stringOfJsonData = '{"name": "Zophie", "isCat": true, "miceCaught": 0,
   "felineIQ": null}'
>>> import json
>>> jsonDataAsPythonValue = json.loads(stringOfJsonData)
>>> jsonDataAsPythonValue
{'isCat': True, 'miceCaught': 0, 'name': 'Zophie', 'felineIQ': None}
```

After you import the `json` module, you can call `loads()` and pass it a string of JSON data. Note that JSON strings always use double quotes. It will return that data as a Python dictionary. Python dictionaries are not ordered, so the key-value pairs may appear in a different order when you print `jsonDataAsPythonValue`.

Writing JSON with the `dumps()` Function

The `json.dumps()` function (which means “dump string,” not “dumps”) will translate a Python value into a string of JSON-formatted data. Enter the following into the interactive shell:

```
>>> pythonValue = {'isCat': True, 'miceCaught': 0, 'name': 'Zophie',
'felineIQ': None}
>>> import json
>>> stringOfJsonData = json.dumps(pythonValue)
>>> stringOfJsonData
'{"isCat": true, "felineIQ": null, "miceCaught": 0, "name": "Zophie" }'
```

The value can only be one of the following basic Python data types: dictionary, list, integer, float, string, Boolean, or `None`.

Project: Fetching Current Weather Data

Checking the weather seems fairly trivial: Open your web browser, click the address bar, type the URL to a weather website (or search for one and then click the link), wait for the page to load, look past all the ads, and so on.

Actually, there are a lot of boring steps you could skip if you had a program that downloaded the weather forecast for the next few days and printed it as plaintext. This program uses the `requests` module from Chapter 11 to download data from the Web.

Overall, the program does the following:

- Reads the requested location from the command line.
- Downloads JSON weather data from OpenWeatherMap.org.
- Converts the string of JSON data to a Python data structure.
- Prints the weather for today and the next two days.

So the code will need to do the following:

- Join strings in `sys.argv` to get the location.
- Call `requests.get()` to download the weather data.
- Call `json.loads()` to convert the JSON data to a Python data structure.
- Print the weather forecast.

For this project, open a new file editor window and save it as `quickWeather.py`.

Step 1: Get Location from the Command Line Argument

The input for this program will come from the command line. Make *quickWeather.py* look like this:

```
#! python3
# quickWeather.py - Prints the weather for a location from the command line.

import json, requests, sys

# Compute location from command line arguments.
if len(sys.argv) < 2:
    print('Usage: quickWeather.py location')
    sys.exit()
location = ' '.join(sys.argv[1:])

# TODO: Download the JSON data from OpenWeatherMap.org's API.

# TODO: Load JSON data into a Python variable.
```

In Python, command line arguments are stored in the `sys.argv` list. After the `#!` shebang line and `import` statements, the program will check that there is more than one command line argument. (Recall that `sys.argv` will always have at least one element, `sys.argv[0]`, which contains the Python script's filename.) If there is only one element in the list, then the user didn't provide a location on the command line, and a "usage" message will be provided to the user before the program ends.

Command line arguments are split on spaces. The command line argument `San Francisco, CA` would make `sys.argv` hold `['quickWeather.py', 'San', 'Francisco', 'CA']`. Therefore, call the `join()` method to join all the strings except for the first in `sys.argv`. Store this joined string in a variable named `location`.

Step 2: Download the JSON Data

OpenWeatherMap.org provides real-time weather information in JSON format. Your program simply has to download the page at `http://api.openweathermap.org/data/2.5/forecast/daily?q=<Location>&cnt=3`, where `<Location>` is the name of the city whose weather you want. Add the following to *quickWeather.py*.

```
#! python3
# quickWeather.py - Prints the weather for a location from the command line.

--snip--

# Download the JSON data from OpenWeatherMap.org's API.
url = 'http://api.openweathermap.org/data/2.5/forecast/daily?q=%s&cnt=3' % (location)
response = requests.get(url)
response.raise_for_status()

# TODO: Load JSON data into a Python variable.
```

We have location from our command line arguments. To make the URL we want to access, we use the %s placeholder and insert whatever string is stored in location into that spot in the URL string. We store the result in url and pass url to requests.get(). The requests.get() call returns a Response object, which you can check for errors by calling raise_for_status(). If no exception is raised, the downloaded text will be in response.text.

Step 3: Load JSON Data and Print Weather

The response.text member variable holds a large string of JSON-formatted data. To convert this to a Python value, call the json.loads() function. The JSON data will look something like this:

```
{'city': {'coord': {'lat': 37.7771, 'lon': -122.42},
           'country': 'United States of America',
           'id': '5391959',
           'name': 'San Francisco',
           'population': 0},
 'cnt': 3,
 'cod': '200',
 'list': [{'clouds': 0,
            'deg': 233,
            'dt': 1402344000,
            'humidity': 58,
            'pressure': 1012.23,
            'speed': 1.96,
            'temp': {'day': 302.29,
                      'eve': 296.46,
                      'max': 302.29,
                      'min': 289.77,
                      'morn': 294.59,
                      'night': 289.77},
            'weather': [{"description": "sky is clear",
                         'icon': '01d'}]}]}
```

--snip--

You can see this data by passing weatherData to pprint.pprint(). You may want to check <http://openweathermap.org/> for more documentation on what these fields mean. For example, the online documentation will tell you that the 302.29 after 'day' is the daytime temperature in Kelvin, not Celsius or Fahrenheit.

The weather descriptions you want are after 'main' and 'description'. To neatly print them out, add the following to quickWeather.py.

```
! python3
# quickWeather.py - Prints the weather for a location from the command line.

--snip--

# Load JSON data into a Python variable.
weatherData = json.loads(response.text)
```

```
# Print weather descriptions.  
❶ w = weatherData['list']  
print('Current weather in %s: % (location))  
print(w[0]['weather'][0]['main'], '-', w[0]['weather'][0]['description'])  
print()  
print('Tomorrow: ')  
print(w[1]['weather'][0]['main'], '-', w[1]['weather'][0]['description'])  
print()  
print('Day after tomorrow: ')  
print(w[2]['weather'][0]['main'], '-', w[2]['weather'][0]['description'])
```

Notice how the code stores `weatherData['list']` in the variable `w` to save you some typing ❶. You use `w[0]`, `w[1]`, and `w[2]` to retrieve the dictionaries for today, tomorrow, and the day after tomorrow's weather, respectively. Each of these dictionaries has a `'weather'` key, which contains a list value. You're interested in the first list item, a nested dictionary with several more keys, at index 0. Here, we print the values stored in the `'main'` and `'description'` keys, separated by a hyphen.

When this program is run with the command line argument `quickWeather.py San Francisco, CA`, the output looks something like this:

```
Current weather in San Francisco, CA:  
Clear - sky is clear
```

```
Tomorrow:  
Clouds - few clouds
```

```
Day after tomorrow:  
Clear - sky is clear
```

(The weather is one of the reasons I like living in San Francisco!)

Ideas for Similar Programs

Accessing weather data can form the basis for many types of programs. You can create similar programs to do the following:

- Collect weather forecasts for several campsites or hiking trails to see which one will have the best weather.
- Schedule a program to regularly check the weather and send you a frost alert if you need to move your plants indoors. (Chapter 15 covers scheduling, and Chapter 16 explains how to send email.)
- Pull weather data from multiple sites to show all at once, or calculate and show the average of the multiple weather predictions.

Summary

CSV and JSON are common plaintext formats for storing data. They are easy for programs to parse while still being human readable, so they are often used for simple spreadsheets or web app data. The `csv` and `json` modules greatly simplify the process of reading and writing to CSV and JSON files.

The last few chapters have taught you how to use Python to parse information from a wide variety of file formats. One common task is taking data from a variety of formats and parsing it for the particular information you need. These tasks are often specific to the point that commercial software is not optimally helpful. By writing your own scripts, you can make the computer handle large amounts of data presented in these formats.

In Chapter 15, you'll break away from data formats and learn how to make your programs communicate with you by sending emails and text messages.

Practice Questions

1. What are some features Excel spreadsheets have that CSV spreadsheets don't?
2. What do you pass to `csv.reader()` and `csv.writer()` to create Reader and Writer objects?
3. What modes do `File` objects for reader and writer objects need to be opened in?
4. What method takes a list argument and writes it to a CSV file?
5. What do the `delimiter` and `lineterminator` keyword arguments do?
6. What function takes a string of JSON data and returns a Python data structure?
7. What function takes a Python data structure and returns a string of JSON data?

Practice Project

For practice, write a program that does the following.

Excel-to-CSV Converter

Excel can save a spreadsheet to a CSV file with a few mouse clicks, but if you had to convert hundreds of Excel files to CSVs, it would take hours of clicking. Using the `openpyxl` module from Chapter 12, write a program that reads all the Excel files in the current working directory and outputs them as CSV files.

A single Excel file might contain multiple sheets; you'll have to create one CSV file per *sheet*. The filenames of the CSV files should be *<excel filename>_<sheet title>.csv*, where *<excel filename>* is the filename of the Excel file without the file extension (for example, 'spam_data', not 'spam_data.xlsx') and *<sheet title>* is the string from the `Worksheet` object's `title` variable.

This program will involve many nested for loops. The skeleton of the program will look something like this:

```
for excelFile in os.listdir('.'):
    # Skip non-xlsx files, load the workbook object.
    for sheetName in wb.get_sheet_names():
        # Loop through every sheet in the workbook.
        sheet = wb.get_sheet_by_name(sheetName)

        # Create the CSV filename from the Excel filename and sheet title.
        # Create the csv.writer object for this CSV file.

        # Loop through every row in the sheet.
        for rowNum in range(1, sheet.get_highest_row() + 1):
            rowData = []      # append each cell to this list
            # Loop through each cell in the row.
            for colNum in range(1, sheet.get_highest_column() + 1):
                # Append each cell's data to rowData.

            # Write the rowData list to the CSV file.

    csvFile.close()
```

Download the ZIP file `excelSpreadsheets.zip` from <http://nostarch.com/automatestuff/>, and unzip the spreadsheets into the same directory as your program. You can use these as the files to test the program on.

15

KEEPING TIME, SCHEDULING TASKS, AND LAUNCHING PROGRAMS



Running programs while you’re sitting at your computer is fine, but it’s also useful to have programs run without your direct supervision. Your computer’s clock can schedule programs to run code at some specified time and date or at regular intervals. For example, your program could scrape a website every hour to check for changes or do a CPU-intensive task at 4 AM while you sleep. Python’s `time` and `datetime` modules provide these functions.

You can also write programs that launch other programs on a schedule by using the `subprocess` and `threading` modules. Often, the fastest way to program is to take advantage of applications that other people have already written.

The time Module

Your computer’s system clock is set to a specific date, time, and time zone. The built-in `time` module allows your Python programs to read the system clock for the current time. The `time.time()` and `time.sleep()` functions are the most useful in the `time` module.

The `time.time()` Function

The *Unix epoch* is a time reference commonly used in programming: 12 AM on January 1, 1970, Coordinated Universal Time (UTC). The `time.time()` function returns the number of seconds since that moment as a float value. (Recall that a float is just a number with a decimal point.) This number is called an *epoch timestamp*. For example, enter the following into the interactive shell:

```
>>> import time
>>> time.time()
1425063955.068649
```

Here I’m calling `time.time()` on February 27, 2015, at 11:05 Pacific Standard Time, or 7:05 PM UTC. The return value is how many seconds have passed between the Unix epoch and the moment `time.time()` was called.

NOTE

The interactive shell examples will yield dates and times for when I wrote this chapter in February 2015. Unless you’re a time traveler, your dates and times will be different.

Epoch timestamps can be used to *profile* code, that is, measure how long a piece of code takes to run. If you call `time.time()` at the beginning of the code block you want to measure and again at the end, you can subtract the first timestamp from the second to find the elapsed time between those two calls. For example, open a new file editor window and enter the following program:

```
import time
❶ def calcProd():
    # Calculate the product of the first 100,000 numbers.
    product = 1
    for i in range(1, 100000):
        product = product * i
    return product

❷ startTime = time.time()
prod = calcProd()
❸ endTime = time.time()
❹ print('The result is %s digits long.' % (len(str(prod))))
❺ print('Took %s seconds to calculate.' % (endTime - startTime))
```

At ❶, we define a function `calcProd()` to loop through the integers from 1 to 99,999 and return their product. At ❷, we call `time.time()` and store it in `startTime`. Right after calling `calcProd()`, we call `time.time()` again and store it in `endTime` ❸. We end by printing the length of the product returned by `calcProd()` ❹ and how long it took to run `calcProd()` ❺.

Save this program as `calcProd.py` and run it. The output will look something like this:

```
The result is 456569 digits long.  
Took 2.844162940979004 seconds to calculate.
```

NOTE

Another way to profile your code is to use the `cProfile.run()` function, which provides a much more informative level of detail than the simple `time.time()` technique. The `cProfile.run()` function is explained at <https://docs.python.org/3/library/profile.html>.

The `time.sleep()` Function

If you need to pause your program for a while, call the `time.sleep()` function and pass it the number of seconds you want your program to stay paused. Enter the following into the interactive shell:

```
>>> import time  
>>> for i in range(3):  
❶    print('Tick')  
❷    time.sleep(1)  
❸    print('Tock')  
❹    time.sleep(1)  
  
Tick  
Tock  
Tick  
Tock  
Tick  
Tock  
❺ >>> time.sleep(5)
```

The for loop will print Tick ❶, pause for one second ❷, print Tock ❸, pause for one second ❹, print Tick, pause, and so on until Tick and Tock have each been printed three times.

The `time.sleep()` function will *block*—that is, it will not return and release your program to execute other code—until after the number of seconds you passed to `time.sleep()` has elapsed. For example, if you enter `time.sleep(5)` ❺, you'll see that the next prompt (`>>>`) doesn't appear until five seconds have passed.

Be aware that pressing CTRL-C will not interrupt `time.sleep()` calls in IDLE. IDLE waits until the entire pause is over before raising the `KeyboardInterrupt` exception. To work around this problem, instead of having a single `time.sleep(30)` call to pause for 30 seconds, use a for loop to make 30 calls to `time.sleep(1)`.

```
>>> for i in range(30):
    time.sleep(1)
```

If you press CTRL-C sometime during these 30 seconds, you should see the `KeyboardInterrupt` exception thrown right away.

Rounding Numbers

When working with times, you'll often encounter float values with many digits after the decimal. To make these values easier to work with, you can shorten them with Python's built-in `round()` function, which rounds a float to the precision you specify. Just pass in the number you want to round, plus an optional second argument representing how many digits after the decimal point you want to round it to. If you omit the second argument, `round()` rounds your number to the nearest whole integer. Enter the following into the interactive shell:

```
>>> import time
>>> now = time.time()
>>> now
1425064108.017826
>>> round(now, 2)
1425064108.02
>>> round(now, 4)
1425064108.0178
>>> round(now)
1425064108
```

After importing `time` and storing `time.time()` in `now`, we call `round(now, 2)` to round `now` to two digits after the decimal, `round(now, 4)` to round to four digits after the decimal, and `round(now)` to round to the nearest integer.

Project: Super Stopwatch

Say you want to track how much time you spend on boring tasks you haven't automated yet. You don't have a physical stopwatch, and it's surprisingly difficult to find a free stopwatch app for your laptop or smartphone that isn't covered in ads and doesn't send a copy of your browser history to marketers. (It says it can do this in the license agreement you agreed to. You did read the license agreement, didn't you?) You can write a simple stopwatch program yourself in Python.

At a high level, here's what your program will do:

- Track the amount of time elapsed between presses of the ENTER key, with each key press starting a new "lap" on the timer.
- Print the lap number, total time, and lap time.

This means your code will need to do the following:

- Find the current time by calling `time.time()` and store it as a timestamp at the start of the program, as well as at the start of each lap.
- Keep a lap counter and increment it every time the user presses ENTER.
- Calculate the elapsed time by subtracting timestamps.
- Handle the `KeyboardInterrupt` exception so the user can press CTRL-C to quit.

Open a new file editor window and save it as `stopwatch.py`.

Step 1: Set Up the Program to Track Times

The stopwatch program will need to use the current time, so you'll want to import the `time` module. Your program should also print some brief instructions to the user before calling `input()`, so the timer can begin after the user presses ENTER. Then the code will start tracking lap times.

Enter the following code into the file editor, writing a TODO comment as a placeholder for the rest of the code:

```
#! python3
# stopwatch.py - A simple stopwatch program.

import time

# Display the program's instructions.
print('Press ENTER to begin. Afterwards, press ENTER to "click" the stopwatch.
Press Ctrl-C to quit.')
input()                      # press Enter to begin
print('Started.')
startTime = time.time()      # get the first lap's start time
lastTime = startTime
lapNum = 1

# TODO: Start tracking the lap times.
```

Now that we've written the code to display the instructions, start the first lap, note the time, and set our lap count to 1.

Step 2: Track and Print Lap Times

Now let's write the code to start each new lap, calculate how long the previous lap took, and calculate the total time elapsed since starting the stopwatch. We'll display the lap time and total time and increase the lap count for each new lap. Add the following code to your program:

```
#! python3
# stopwatch.py - A simple stopwatch program.

import time

--snip--
```

```
# Start tracking the lap times.  
❶ try:  
❷     while True:  
❸         input()  
❹         lapTime = round(time.time() - lastTime, 2)  
❺         totalTime = round(time.time() - startTime, 2)  
❻         print('Lap #%s: %s (%s)' % (lapNum, totalTime, lapTime), end='')  
❼         lapNum += 1  
➋         lastTime = time.time() # reset the last lap time  
❾ except KeyboardInterrupt:  
    # Handle the Ctrl-C exception to keep its error message from displaying.  
    print('\nDone.')
```

If the user presses CTRL-C to stop the stopwatch, the `KeyboardInterrupt` exception will be raised, and the program will crash if its execution is not a `try` statement. To prevent crashing, we wrap this part of the program in a `try` statement ❶. We'll handle the exception in the `except` clause ❾, so when CTRL-C is pressed and the exception is raised, the program execution moves to the `except` clause to print `Done`, instead of the `KeyboardInterrupt` error message. Until this happens, the execution is inside an infinite loop ❷ that calls `input()` and waits until the user presses ENTER to end a lap. When a lap ends, we calculate how long the lap took by subtracting the start time of the lap, `lastTime`, from the current time, `time.time()` ❸. We calculate the total time elapsed by subtracting the overall start time of the stopwatch, `startTime`, from the current time ❹.

Since the results of these time calculations will have many digits after the decimal point (such as `4.766272783279419`), we use the `round()` function to round the float value to two digits at ❽ and ❾.

At ❻, we print the lap number, total time elapsed, and the lap time. Since the user pressing ENTER for the `input()` call will print a newline to the screen, pass `end=''` to the `print()` function to avoid double-spacing the output. After printing the lap information, we get ready for the next lap by adding 1 to the count `lapNum` and setting `lastTime` to the current time, which is the start time of the next lap.

Ideas for Similar Programs

Time tracking opens up several possibilities for your programs. Although you can download apps to do some of these things, the benefit of writing programs yourself is that they will be free and not bloated with ads and useless features. You could write similar programs to do the following:

- Create a simple timesheet app that records when you type a person's name and uses the current time to clock them in or out.
- Add a feature to your program to display the elapsed time since a process started, such as a download that uses the `requests` module. (See Chapter 11.)
- Intermittently check how long a program has been running and offer the user a chance to cancel tasks that are taking too long.

The `datetime` Module

The `time` module is useful for getting a Unix epoch timestamp to work with. But if you want to display a date in a more convenient format, or do arithmetic with dates (for example, figuring out what date was 205 days ago or what date is 123 days from now), you should use the `datetime` module.

The `datetime` module has its own `datetime` data type. `datetime` values represent a specific moment in time. Enter the following into the interactive shell:

```
>>> import datetime
❶ >>> datetime.datetime.now()
❷ datetime.datetime(2015, 2, 27, 11, 10, 49, 55, 53)
❸ >>> dt = datetime.datetime(2015, 10, 21, 16, 29, 0)
❹ >>> dt.year, dt.month, dt.day
(2015, 10, 21)
❺ >>> dt.hour, dt.minute, dt.second
(16, 29, 0)
```

Calling `datetime.datetime.now()` ❶ returns a `datetime` object ❷ for the current date and time, according to your computer’s clock. This object includes the year, month, day, hour, minute, second, and microsecond of the current moment. You can also retrieve a `datetime` object for a specific moment by using the `datetime.datetime()` function ❸, passing it integers representing the year, month, day, hour, and second of the moment you want. These integers will be stored in the `datetime` object’s `year`, `month`, `day` ❹, `hour`, `minute`, and `second` ❺ attributes.

A Unix epoch timestamp can be converted to a `datetime` object with the `datetime.datetime.fromtimestamp()` function. The date and time of the `datetime` object will be converted for the local time zone. Enter the following into the interactive shell:

```
>>> datetime.datetime.fromtimestamp(1000000)
datetime.datetime(1970, 1, 12, 5, 46, 40)
>>> datetime.datetime.fromtimestamp(time.time())
datetime.datetime(2015, 2, 27, 11, 13, 0, 604980)
```

Calling `datetime.datetime.fromtimestamp()` and passing it 1000000 returns a `datetime` object for the moment 1,000,000 seconds after the Unix epoch. Passing `time.time()`, the Unix epoch timestamp for the current moment, returns a `datetime` object for the current moment. So the expressions `datetime.datetime.now()` and `datetime.datetime.fromtimestamp(time.time())` do the same thing; they both give you a `datetime` object for the present moment.

NOTE

These examples were entered on a computer set to Pacific Standard Time. If you’re in another time zone, your results will look different.

`datetime` objects can be compared with each other using comparison operators to find out which one precedes the other. The later `datetime` object is the “greater” value. Enter the following into the interactive shell:

```
❶ >>> halloween2015 = datetime.datetime(2015, 10, 31, 0, 0, 0)
❷ >>> newyears2016 = datetime.datetime(2016, 1, 1, 0, 0, 0)
    >>> oct31_2015 = datetime.datetime(2015, 10, 31, 0, 0, 0)
❸ >>> halloween2015 == oct31_2015
    True
❹ >>> halloween2015 > newyears2016
    False
❺ >>> newyears2016 > halloween2015
    True
    >>> newyears2016 != oct31_2015
    True
```

Make a `datetime` object for the first moment (midnight) of October 31, 2015 and store it in `halloween2015` ❶. Make a `datetime` object for the first moment of January 1, 2016 and store it in `newyears2016` ❷. Then make another object for midnight on October 31, 2015 and store it in `oct31_2015`. Comparing `halloween2015` and `oct31_2015` shows that they’re equal ❸. Comparing `newyears2016` and `halloween2015` shows that `newyears2016` is greater (later) than `halloween2015` ❹ ❺.

The `timedelta` Data Type

The `datetime` module also provides a `timedelta` data type, which represents a *duration* of time rather than a *moment* in time. Enter the following into the interactive shell:

```
❶ >>> delta = datetime.timedelta(days=11, hours=10, minutes=9, seconds=8)
❷ >>> delta.days, delta.seconds, delta.microseconds
(11, 36548, 0)
>>> delta.total_seconds()
986948.0
>>> str(delta)
'11 days, 10:09:08'
```

To create a `timedelta` object, use the `datetime.timedelta()` function. The `datetime.timedelta()` function takes keyword arguments `weeks`, `days`, `hours`, `minutes`, `seconds`, `milliseconds`, and `microseconds`. There is no `month` or `year` keyword argument because “a month” or “a year” is a variable amount of time depending on the particular month or year. A `timedelta` object has the total duration represented in days, seconds, and microseconds. These numbers are stored in the `days`, `seconds`, and `microseconds` attributes, respectively. The `total_seconds()` method will return the duration in number of seconds alone. Passing a `timedelta` object to `str()` will return a nicely formatted, human-readable string representation of the object.

In this example, we pass keyword arguments to `datetime.timedelta()` to specify a duration of 11 days, 10 hours, 9 minutes, and 8 seconds, and store the returned `timedelta` object in `delta` ❶. This `timedelta` object's `days` attribute stores 11, and its `seconds` attribute stores 36548 (10 hours, 9 minutes, and 8 seconds, expressed in seconds) ❷. Calling `total_seconds()` tells us that 11 days, 10 hours, 9 minutes, and 8 seconds is 986,948 seconds. Finally, passing the `timedelta` object to `str()` returns a string clearly explaining the duration.

The arithmetic operators can be used to perform *date arithmetic* on `datetime` values. For example, to calculate the date 1,000 days from now, enter the following into the interactive shell:

```
>>> dt = datetime.datetime.now()
>>> dt
datetime.datetime(2015, 2, 27, 18, 38, 50, 636181)
>>> thousandDays = datetime.timedelta(days=1000)
>>> dt + thousandDays
datetime.datetime(2017, 11, 23, 18, 38, 50, 636181)
```

First, make a `datetime` object for the current moment and store it in `dt`. Then make a `timedelta` object for a duration of 1,000 days and store it in `thousandDays`. Add `dt` and `thousandDays` together to get a `datetime` object for the date 1,000 days from now. Python will do the date arithmetic to figure out that 1,000 days after February 27, 2015, will be November 23, 2017. This is useful because when you calculate 1,000 days from a given date, you have to remember how many days are in each month and factor in leap years and other tricky details. The `datetime` module handles all of this for you.

`timedelta` objects can be added or subtracted with `datetime` objects or other `timedelta` objects using the `+` and `-` operators. A `timedelta` object can be multiplied or divided by integer or float values with the `*` and `/` operators. Enter the following into the interactive shell:

```
❶ >>> oct21st = datetime.datetime(2015, 10, 21, 16, 29, 0)
❷ >>> aboutThirtyYears = datetime.timedelta(days=365 * 30)
>>> oct21st
datetime.datetime(2015, 10, 21, 16, 29)
>>> oct21st - aboutThirtyYears
datetime.datetime(1985, 10, 28, 16, 29)
>>> oct21st - (2 * aboutThirtyYears)
datetime.datetime(1955, 11, 5, 16, 29)
```

Here we make a `datetime` object for October 21, 2015 ❶ and a `timedelta` object for a duration of about 30 years (we're assuming 365 days for each of those years) ❷. Subtracting `aboutThirtyYears` from `oct21st` gives us a `datetime` object for the date 30 years before October 21, 2015. Subtracting `2 * aboutThirtyYears` from `oct21st` returns a `datetime` object for the date 60 years before October 21, 2015.

Pausing Until a Specific Date

The `time.sleep()` method lets you pause a program for a certain number of seconds. By using a `while` loop, you can pause your programs until a specific date. For example, the following code will continue to loop until Halloween 2016:

```
import datetime
import time
halloween2016 = datetime.datetime(2016, 10, 31, 0, 0, 0)
while datetime.datetime.now() < halloween2016:
    time.sleep(1)
```

The `time.sleep(1)` call will pause your Python program so that the computer doesn't waste CPU processing cycles simply checking the time over and over. Rather, the `while` loop will just check the condition once per second and continue with the rest of the program after Halloween 2016 (or whenever you program it to stop).

Converting `datetime` Objects into Strings

Epoch timestamps and `datetime` objects aren't very friendly to the human eye. Use the `strftime()` method to display a `datetime` object as a string. (The `f` in the name of the `strftime()` function stands for *format*.)

The `strftime()` method uses directives similar to Python's string formatting. Table 15-1 has a full list of `strftime()` directives.

Table 15-1: `strftime()` Directives

strftime directive	Meaning
<code>%Y</code>	Year with century, as in '2014'
<code>%y</code>	Year without century, '00' to '99' (1970 to 2069)
<code>%m</code>	Month as a decimal number, '01' to '12'
<code>%B</code>	Full month name, as in 'November'
<code>%b</code>	Abbreviated month name, as in 'Nov'
<code>%d</code>	Day of the month, '01' to '31'
<code>%j</code>	Day of the year, '001' to '366'
<code>%w</code>	Day of the week, '0' (Sunday) to '6' (Saturday)
<code>%A</code>	Full weekday name, as in 'Monday'
<code>%a</code>	Abbreviated weekday name, as in 'Mon'
<code>%H</code>	Hour (24-hour clock), '00' to '23'
<code>%I</code>	Hour (12-hour clock), '01' to '12'
<code>%M</code>	Minute, '00' to '59'
<code>%S</code>	Second, '00' to '59'
<code>%p</code>	'AM' or 'PM'
<code>%%</code>	Literal '%' character

Pass `strrftime()` a custom format string containing formatting directives (along with any desired slashes, colons, and so on), and `strrftime()` will return the `datetime` object's information as a formatted string. Enter the following into the interactive shell:

```
>>> oct21st = datetime.datetime(2015, 10, 21, 16, 29, 0)
>>> oct21st.strftime('%Y/%m/%d %H:%M:%S')
'2015/10/21 16:29:00'
>>> oct21st.strftime('%I:%M %p')
'04:29 PM'
>>> oct21st.strftime("%B of '%y")
"October of '15"
```

Here we have a `datetime` object for October 21, 2015 at 4:29 PM, stored in `oct21st`. Passing `strftime()` the custom format string '`%Y/%m/%d %H:%M:%S`' returns a string containing 2015, 10, and 21 separated by slashes and 16, 29, and 00 separated by colons. Passing '`%I:%M% p`' returns '`04:29 PM`', and passing "`%B of '%y`" returns "`October of '15`". Note that `strftime()` doesn't begin with `datetime.datetime`.

Converting Strings into datetime Objects

If you have a string of date information, such as '`2015/10/21 16:29:00`' or '`October 21, 2015`', and need to convert it to a `datetime` object, use the `datetime.datetime.strptime()` function. The `strptime()` function is the inverse of the `strftime()` method. A custom format string using the same directives as `strftime()` must be passed so that `strptime()` knows how to parse and understand the string. (The *p* in the name of the `strptime()` function stands for *parse*.)

Enter the following into the interactive shell:

```
❶ >>> datetime.datetime.strptime('October 21, 2015', '%B %d, %Y')
datetime.datetime(2015, 10, 21, 0, 0)
>>> datetime.datetime.strptime('2015/10/21 16:29:00', '%Y/%m/%d %H:%M:%S')
datetime.datetime(2015, 10, 21, 16, 29)
>>> datetime.datetime.strptime("October of '15", "%B of '%y")
datetime.datetime(2015, 10, 1, 0, 0)
>>> datetime.datetime.strptime("November of '63", "%B of '%y")
datetime.datetime(2063, 11, 1, 0, 0)
```

To get a `datetime` object from the string '`October 21, 2015`', pass '`October 21, 2015`' as the first argument to `strptime()` and the custom format string that corresponds to '`October 21, 2015`' as the second argument ❶. The string with the date information must match the custom format string exactly, or Python will raise a `ValueError` exception.

Review of Python’s Time Functions

Dates and times in Python can involve quite a few different data types and functions. Here’s a review of the three different types of values used to represent time:

- A Unix epoch timestamp (used by the `time` module) is a float or integer value of the number of seconds since 12 AM on January 1, 1970, UTC.
- A `datetime` object (of the `datetime` module) has integers stored in the attributes `year`, `month`, `day`, `hour`, `minute`, and `second`.
- A `timedelta` object (of the `datetime` module) represents a time duration, rather than a specific moment.

Here’s a review of time functions and their parameters and return values:

- The `time.time()` function returns an epoch timestamp float value of the current moment.
- The `time.sleep(seconds)` function stops the program for the amount of seconds specified by the `seconds` argument.
- The `datetime.datetime(year, month, day, hour, minute, second)` function returns a `datetime` object of the moment specified by the arguments. If `hour`, `minute`, or `second` arguments are not provided, they default to 0.
- The `datetime.datetime.now()` function returns a `datetime` object of the current moment.
- The `datetime.datetime.fromtimestamp(epoch)` function returns a `datetime` object of the moment represented by the `epoch` timestamp argument.
- The `datetime.timedelta(weeks, days, hours, minutes, seconds, milliseconds, microseconds)` function returns a `timedelta` object representing a duration of time. The function’s keyword arguments are all optional and do not include `month` or `year`.
- The `total_seconds()` method for `timedelta` objects returns the number of seconds the `timedelta` object represents.
- The `strftime(format)` method returns a string of the time represented by the `datetime` object in a custom format that’s based on the `format` string. See Table 15-1 for the format details.
- The `datetime.datetime.strptime(time_string, format)` function returns a `datetime` object of the moment specified by `time_string`, parsed using the `format` string argument. See Table 15-1 for the format details.

Multithreading

To introduce the concept of multithreading, let's look at an example situation. Say you want to schedule some code to run after a delay or at a specific time. You could add code like the following at the start of your program:

```
import time, datetime

startTime = datetime.datetime(2029, 10, 31, 0, 0, 0)
while datetime.datetime.now() < startTime:
    time.sleep(1)

print('Program now starting on Halloween 2029')
--snip--
```

This code designates a start time of October 31, 2029, and keeps calling `time.sleep(1)` until the start time arrives. Your program cannot do anything while waiting for the loop of `time.sleep()` calls to finish; it just sits around until Halloween 2029. This is because Python programs by default have a single *thread* of execution.

To understand what a thread of execution is, remember the Chapter 2 discussion of flow control, when you imagined the execution of a program as placing your finger on a line of code in your program and moving to the next line or wherever it was sent by a flow control statement. A *single-threaded* program has only one finger. But a *multithreaded* program has multiple fingers. Each finger still moves to the next line of code as defined by the flow control statements, but the fingers can be at different places in the program, executing different lines of code at the same time. (All of the programs in this book so far have been single threaded.)

Rather than having all of your code wait until the `time.sleep()` function finishes, you can execute the delayed or scheduled code in a separate thread using Python's `threading` module. The separate thread will pause for the `time.sleep` calls. Meanwhile, your program can do other work in the original thread.

To make a separate thread, you first need to make a `Thread` object by calling the `threading.Thread()` function. Enter the following code in a new file and save it as `threadDemo.py`:

```
import threading, time
print('Start of program.')

❶ def takeANap():
    time.sleep(5)
    print('Wake up!')

❷ threadObj = threading.Thread(target=takeANap)
❸ threadObj.start()

print('End of program.'
```

At ❶, we define a function that we want to use in a new thread. To create a Thread object, we call `threading.Thread()` and pass it the keyword argument `target=takeANap` ❷. This means the function we want to call in the new thread is `takeANap()`. Notice that the keyword argument is `target=takeANap`, not `target=takeANap()`. This is because you want to pass the `takeANap()` function itself as the argument, not call `takeANap()` and pass its return value.

After we store the Thread object created by `threading.Thread()` in `threadObj`, we call `threadObj.start()` ❸ to create the new thread and start executing the target function in the new thread. When this program is run, the output will look like this:

```
Start of program.  
End of program.  
Wake up!
```

This can be a bit confusing. If `print('End of program.')` is the last line of the program, you might think that it should be the last thing printed. The reason `Wake up!` comes after it is that when `threadObj.start()` is called, the target function for `threadObj` is run in a new thread of execution. Think of it as a second finger appearing at the start of the `takeANap()` function. The main thread continues to `print('End of program.')`. Meanwhile, the new thread that has been executing the `time.sleep(5)` call, pauses for 5 seconds. After it wakes from its 5-second nap, it prints '`Wake up!`' and then returns from the `takeANap()` function. Chronologically, '`Wake up!`' is the last thing printed by the program.

Normally a program terminates when the last line of code in the file has run (or the `sys.exit()` function is called). But `threadDemo.py` has two threads. The first is the original thread that began at the start of the program and ends after `print('End of program.')`. The second thread is created when `threadObj.start()` is called, begins at the start of the `takeANap()` function, and ends after `takeANap()` returns.

A Python program will not terminate until all its threads have terminated. When you ran `threadDemo.py`, even though the original thread had terminated, the second thread was still executing the `time.sleep(5)` call.

Passing Arguments to the Thread's Target Function

If the target function you want to run in the new thread takes arguments, you can pass the target function's arguments to `threading.Thread()`. For example, say you wanted to run this `print()` call in its own thread:

```
>>> print('Cats', 'Dogs', 'Frogs', sep=' & ')  
Cats & Dogs & Frogs
```

This `print()` call has three regular arguments, '`Cats`', '`Dogs`', and '`Frogs`', and one keyword argument, `sep=' & '`. The regular arguments can be passed

as a list to the args keyword argument in `threading.Thread()`. The keyword argument can be specified as a dictionary to the kwargs keyword argument in `threading.Thread()`.

Enter the following into the interactive shell:

```
>>> import threading  
>>> threadObj = threading.Thread(target=print, args=['Cats', 'Dogs', 'Frogs'],  
    kwargs={'sep': ' & '})  
>>> threadObj.start()  
Cats & Dogs & Frogs
```

To make sure the arguments 'Cats', 'Dogs', and 'Frogs' get passed to `print()` in the new thread, we pass `args=['Cats', 'Dogs', 'Frogs']` to `threading.Thread()`. To make sure the keyword argument `sep=' & '` gets passed to `print()` in the new thread, we pass `kwargs={'sep': ' & '}` to `threading.Thread()`.

The `threadObj.start()` call will create a new thread to call the `print()` function, and it will pass 'Cats', 'Dogs', and 'Frogs' as arguments and ' & ' for the `sep` keyword argument.

This is an incorrect way to create the new thread that calls `print()`:

```
threadObj = threading.Thread(target=print('Cats', 'Dogs', 'Frogs', sep=' & '))
```

What this ends up doing is calling the `print()` function and passing its return value (`print()`'s return value is always `None`) as the target keyword argument. It *doesn't* pass the `print()` function itself. When passing arguments to a function in a new thread, use the `threading.Thread()` function's `args` and `kwargs` keyword arguments.

Concurrency Issues

You can easily create several new threads and have them all running at the same time. But multiple threads can also cause problems called *concurrency issues*. These issues happen when threads read and write variables at the same time, causing the threads to trip over each other. Concurrency issues can be hard to reproduce consistently, making them hard to debug.

Multithreaded programming is its own wide subject and beyond the scope of this book. What you have to keep in mind is this: To avoid concurrency issues, never let multiple threads read or write the same variables. When you create a new `Thread` object, make sure its target function uses only local variables in that function. This will avoid hard-to-debug concurrency issues in your programs.

NOTE

A beginner's tutorial on multithreaded programming is available at <http://nostarch.com/automatestuff/>.

Project: Multithreaded XKCD Downloader

In Chapter 11, you wrote a program that downloaded all of the XKCD comic strips from the XKCD website. This was a single-threaded program: It downloaded one comic at a time. Much of the program’s running time was spent establishing the network connection to begin the download and writing the downloaded images to the hard drive. If you have a broadband Internet connection, your single-threaded program wasn’t fully utilizing the available bandwidth.

A multithreaded program that has some threads downloading comics while others are establishing connections and writing the comic image files to disk uses your Internet connection more efficiently and downloads the collection of comics more quickly. Open a new file editor window and save it as *multidownloadXkcd.py*. You will modify this program to add multithreading. The completely modified source code is available to download from <http://nostarch.com/automatestuff/>.

Step 1: Modify the Program to Use a Function

This program will mostly be the same downloading code from Chapter 11, so I’ll skip the explanation for the Requests and BeautifulSoup code. The main changes you need to make are importing the `threading` module and making a `downloadXkcd()` function, which takes starting and ending comic numbers as parameters.

For example, calling `downloadXkcd(140, 280)` would loop over the downloading code to download the comics at <http://xkcd.com/140>, <http://xkcd.com/141>, <http://xkcd.com/142>, and so on, up to <http://xkcd.com/279>. Each thread that you create will call `downloadXkcd()` and pass a different range of comics to download.

Add the following code to your *multidownloadXkcd.py* program:

```
#! python3
# multidownloadXkcd.py - Downloads XKCD comics using multiple threads.

import requests, os, bs4, threading
❶ os.makedirs('xkcd', exist_ok=True)      # store comics in ./xkcd

❷ def downloadXkcd(startComic, endComic):
❸     for urlNumber in range(startComic, endComic):
❹         # Download the page.
❺         print('Downloading page http://xkcd.com/%s...' % (urlNumber))
❻         res = requests.get('http://xkcd.com/%s' % (urlNumber))
❽         res.raise_for_status()

❾         soup = bs4.BeautifulSoup(res.text)

❿         # Find the URL of the comic image.
❾         comicElem = soup.select('#comic img')
❿         if comicElem == []:
             print('Could not find comic image.')
         else:
             comicUrl = comicElem[0].get('src')
             # Download the image.
             imgRes = requests.get(comicUrl)
             imgRes.raise_for_status()
             imgFile = open(os.path.join('xkcd', str(urlNumber) + '_comic.jpg'), 'wb')
             for chunk in imgRes.iter_content(100000):
                 imgFile.write(chunk)
             imgFile.close()
```

```

        else:
    ⑦     comicUrl = comicElem[0].get('src')
          # Download the image.
          print('Downloading image %s...' % (comicUrl))
    ⑧     res = requests.get(comicUrl)
          res.raise_for_status()

          # Save the image to ./xkcd.
          imageFile = open(os.path.join('xkcd', os.path.basename(comicUrl)), 'wb')
          for chunk in res.iter_content(100000):
              imageFile.write(chunk)
          imageFile.close()

# TODO: Create and start the Thread objects.
# TODO: Wait for all threads to end.

```

After importing the modules we need, we make a directory to store comics in ① and start defining `downloadXkcd()` ②. We loop through all the numbers in the specified range ③ and download each page ④. We use Beautiful Soup to look through the HTML of each page ⑤ and find the comic image ⑥. If no comic image is found on a page, we print a message. Otherwise, we get the URL of the image ⑦ and download the image ⑧. Finally, we save the image to the directory we created.

Step 2: Create and Start Threads

Now that we've defined `downloadXkcd()`, we'll create the multiple threads that each call `downloadXkcd()` to download different ranges of comics from the XKCD website. Add the following code to *multidownloadXkcd.py* after the `downloadXkcd()` function definition:

```

#!/usr/bin/python3
# multidownloadXkcd.py - Downloads XKCD comics using multiple threads.

--snip--

# Create and start the Thread objects.
downloadThreads = []                  # a list of all the Thread objects
for i in range(0, 1400, 100):         # loops 14 times, creates 14 threads
    downloadThread = threading.Thread(target=downloadXkcd, args=(i, i + 99))
    downloadThreads.append(downloadThread)
    downloadThread.start()

```

First we make an empty list `downloadThreads`; the list will help us keep track of the many Thread objects we'll create. Then we start our for loop. Each time through the loop, we create a Thread object with `threading.Thread()`, append the Thread object to the list, and call `start()` to start running `downloadXkcd()` in the new thread. Since the for loop sets the `i` variable from 0 to 1400 at steps of 100, `i` will be set to 0 on the first iteration, 100 on the second iteration, 200 on the third, and so on. Since we pass `args=(i, i + 99)` to `threading.Thread()`, the two arguments passed to `downloadXkcd()` will be 0 and 99 on the first iteration, 100 and 199 on the second iteration, 200 and 299 on the third, and so on.

As the `Thread` object's `start()` method is called and the new thread begins to run the code inside `downloadXkcd()`, the main thread will continue to the next iteration of the `for` loop and create the next thread.

Step 3: Wait for All Threads to End

The main thread moves on as normal while the other threads we create download comics. But say there's some code you don't want to run in the main thread until all the threads have completed. Calling a `Thread` object's `join()` method will block until that thread has finished. By using a `for` loop to iterate over all the `Thread` objects in the `downloadThreads` list, the main thread can call the `join()` method on each of the other threads. Add the following to the bottom of your program:

```
#! python3
# multidownloadXkcd.py - Downloads XKCD comics using multiple threads.

--snip--

# Wait for all threads to end.
for downloadThread in downloadThreads:
    downloadThread.join()
print('Done.')
```

The `'Done.'` string will not be printed until all of the `join()` calls have returned. If a `Thread` object has already completed when its `join()` method is called, then the method will simply return immediately. If you wanted to extend this program with code that runs only after all of the comics downloaded, you could replace the `print('Done.')` line with your new code.

Launching Other Programs from Python

Your Python program can start other programs on your computer with the `Popen()` function in the built-in `subprocess` module. (The *P* in the name of the `Popen()` function stands for *process*.) If you have multiple instances of an application open, each of those instances is a separate process of the same program. For example, if you open multiple windows of your web browser at the same time, each of those windows is a different process of the web browser program. See Figure 15-1 for an example of multiple calculator processes open at once.

Every process can have multiple threads. Unlike threads, a process cannot directly read and write another process's variables. If you think of a multithreaded program as having multiple fingers following source code, then having multiple processes of the same program open is like having a friend with a separate copy of the program's source code. You are both independently executing the same program.

If you want to start an external program from your Python script, pass the program's filename to `subprocess.Popen()`. (On Windows, right-click the application's **Start** menu item and select **Properties** to view the application's

filename. On OS X, CTRL-click the application and select **Show Package Contents** to find the path to the executable file.) The `Popen()` function will then immediately return. Keep in mind that the launched program is not run in the same thread as your Python program.



Figure 15-1: Six running processes of the same calculator program

On a Windows computer, enter the following into the interactive shell:

```
>>> import subprocess
>>> subprocess.Popen('C:\\Windows\\System32\\calc.exe')
<subprocess.Popen object at 0x000000003055A58>
```

On Ubuntu Linux, you would enter the following:

```
>>> import subprocess
>>> subprocess.Popen('/usr/bin/gnome-calculator')
<subprocess.Popen object at 0x7f2bcf93b20>
```

On OS X, the process is slightly different. See “Opening Files with Default Applications” on page 355.

The return value is a `Popen` object, which has two useful methods: `poll()` and `wait()`.

You can think of the `poll()` method as asking your friend if she’s finished running the code you gave her. The `poll()` method will return `None` if the process is still running at the time `poll()` is called. If the program has terminated, it will return the process’s integer *exit code*. An exit code is used

to indicate whether the process terminated without errors (an exit code of 0) or whether an error caused the process to terminate (a nonzero exit code—generally 1, but it may vary depending on the program).

The `wait()` method is like waiting for your friend to finish working on her code before you keep working on yours. The `wait()` method will block until the launched process has terminated. This is helpful if you want your program to pause until the user finishes with the other program. The return value of `wait()` is the process's integer exit code.

On Windows, enter the following into the interactive shell. Note that the `wait()` call will block until you quit the launched calculator program.

```
❶ >>> calcProc = subprocess.Popen('c:\\Windows\\System32\\calc.exe')
❷ >>> calcProc.poll() == None
True
❸ >>> calcProc.wait()
0
>>> calcProc.poll()
0
```

Here we open a calculator process ❶. While it's still running, we check if `poll()` returns `None` ❷. It should, as the process is still running. Then we close the calculator program and call `wait()` on the terminated process ❸. `wait()` and `poll()` now return 0, indicating that the process terminated without errors.

Passing Command Line Arguments to Popen()

You can pass command line arguments to processes you create with `Popen()`. To do so, you pass a list as the sole argument to `Popen()`. The first string in this list will be the executable filename of the program you want to launch; all the subsequent strings will be the command line arguments to pass to the program when it starts. In effect, this list will be the value of `sys.argv` for the launched program.

Most applications with a graphical user interface (GUI) don't use command line arguments as extensively as command line-based or terminal-based programs do. But most GUI applications will accept a single argument for a file that the applications will immediately open when they start. For example, if you're using Windows, create a simple text file called `C:\hello.txt` and then enter the following into the interactive shell:

```
>>> subprocess.Popen(['C:\\Windows\\notepad.exe', 'C:\\\\hello.txt'])
<subprocess.Popen object at 0x000000000032DCEB>
```

This will not only launch the Notepad application but also have it immediately open the `C:\hello.txt` file.

Task Scheduler, launchd, and cron

If you are computer savvy, you may know about Task Scheduler on Windows, launchd on OS X, or the cron scheduler on Linux. These well-documented

and reliable tools all allow you to schedule applications to launch at specific times. If you'd like to learn more about them, you can find links to tutorials at <http://nostarch.com/automatestuff/>.

Using your operating system's built-in scheduler saves you from writing your own clock-checking code to schedule your programs. However, use the `time.sleep()` function if you just need your program to pause briefly. Or instead of using the operating system's scheduler, your code can loop until a certain date and time, calling `time.sleep(1)` each time through the loop.

Opening Websites with Python

The `webbrowser.open()` function can launch a web browser from your program to a specific website, rather than opening the browser application with `subprocess.Popen()`. See “Project: *mapIt.py* with the `webbrowser` Module” on page 234 for more details.

Running Other Python Scripts

You can launch a Python script from Python just like any other application. You just have to pass the `python.exe` executable to `Popen()` and the filename of the `.py` script you want to run as its argument. For example, the following would run the `hello.py` script from Chapter 1:

```
>>> subprocess.Popen(['C:\\\\python34\\\\python.exe', 'hello.py'])
<subprocess.Popen object at 0x000000000331CF28>
```

Pass `Popen()` a list containing a string of the Python executable's path and a string of the script's filename. If the script you're launching needs command line arguments, add them to the list after the script's filename. The location of the Python executable on Windows is `C:\\python34\\python.exe`. On OS X, it is `/Library/Frameworks/Python.framework/Versions/3.3/bin/python3`. On Linux, it is `/usr/bin/python3`.

Unlike importing the Python program as a module, when your Python program launches another Python program, the two are run in separate processes and will not be able to share each other's variables.

Opening Files with Default Applications

Double-clicking a `.txt` file on your computer will automatically launch the application associated with the `.txt` file extension. Your computer will have several of these file extension associations set up already. Python can also open files this way with `Popen()`.

Each operating system has a program that performs the equivalent of double-clicking a document file to open it. On Windows, this is the `start` program. On OS X, this is the `open` program. On Ubuntu Linux, this is the `see` program. Enter the following into the interactive shell, passing '`start`', '`open`', or '`see`' to `Popen()` depending on your system:

```
>>> fileObj = open('hello.txt', 'w')
>>> fileObj.write('Hello world!')
```

```
>>> fileObj.close()
>>> import subprocess
>>> subprocess.Popen(['start', 'hello.txt'], shell=True)
```

Here we write Hello world! to a new *hello.txt* file. Then we call `Popen()`, passing it a list containing the program name (in this example, 'start' for Windows) and the filename. We also pass the `shell=True` keyword argument, which is needed only on Windows. The operating system knows all of the file associations and can figure out that it should launch, say, *Notepad.exe* to handle the *hello.txt* file.

On OS X, the `open` program is used for opening both document files and programs. Enter the following into the interactive shell if you have a Mac:

```
>>> subprocess.Popen(['open', '/Applications/Calculator.app/'])
<subprocess.Popen object at 0x10202ff98>
```

The Calculator app should open.

THE UNIX PHILOSOPHY

Programs well designed to be launched by other programs become more powerful than their code alone. The *Unix philosophy* is a set of software design principles established by the programmers of the Unix operating system (on which the modern Linux and OS X are built). It says that it's better to write small, limited-purpose programs that can interoperate, rather than large, feature-rich applications. The smaller programs are easier to understand, and by being interoperable, they can be the building blocks of much more powerful applications.

Smartphone apps follow this approach as well. If your restaurant app needs to display directions to a café, the developers didn't reinvent the wheel by writing their own map code. The restaurant app simply launches a map app while passing it the café's address, just as your Python code would call a function and pass it arguments.

The Python programs you've been writing in this book mostly fit the Unix philosophy, especially in one important way: They use command line arguments rather than `input()` function calls. If all the information your program needs can be supplied up front, it is preferable to have this information passed as command line arguments rather than waiting for the user to type it in. This way, the command line arguments can be entered by a human user or supplied by another program. This interoperable approach will make your programs reusable as part of another program.

The sole exception is that you don't want passwords passed as command line arguments, since the command line may record them as part of its command history feature. Instead, your program should call the `input()` function when it needs you to enter a password.

You can read more about Unix philosophy at https://en.wikipedia.org/wiki/Unix_philosophy/.

Project: Simple Countdown Program

Just like it's hard to find a simple stopwatch application, it can be hard to find a simple countdown application. Let's write a countdown program that plays an alarm at the end of the countdown.

At a high level, here's what your program will do:

- Count down from 60.
- Play a sound file (*alarm.wav*) when the countdown reaches zero.

This means your code will need to do the following:

- Pause for one second in between displaying each number in the countdown by calling `time.sleep()`.
- Call `subprocess.Popen()` to open the sound file with the default application.

Open a new file editor window and save it as *countdown.py*.

Step 1: Count Down

This program will require the `time` module for the `time.sleep()` function and the `subprocess` module for the `subprocess.Popen()` function. Enter the following code and save the file as *countdown.py*:

```
#! python3
# countdown.py - A simple countdown script.

import time, subprocess

❶ timeLeft = 60
while timeLeft > 0:
❷     print(timeLeft, end='')
❸     time.sleep(1)
❹     timeLeft = timeLeft - 1

# TODO: At the end of the countdown, play a sound file.
```

After importing `time` and `subprocess`, make a variable called `timeLeft` to hold the number of seconds left in the countdown ❶. It can start at 60—or you can change the value here to whatever you need or even have it get set from a command line argument.

In a while loop, you display the remaining count ❷, pause for one second ❸, and then decrement the `timeLeft` variable ❹ before the loop starts over again. The loop will keep looping as long as `timeLeft` is greater than 0. After that, the countdown will be over.

Step 2: Play the Sound File

While there are third-party modules to play sound files of various formats, the quick and easy way is to just launch whatever application the user already

uses to play sound files. The operating system will figure out from the `.wav` file extension which application it should launch to play the file. This `.wav` file could easily be some other sound file format, such as `.mp3` or `.ogg`.

You can use any sound file that is on your computer to play at the end of the countdown, or you can download `alarm.wav` from <http://nostarch.com/automatestuff/>.

Add the following to your code:

```
#! python3
# countdown.py - A simple countdown script.

import time, subprocess

--snip--

# At the end of the countdown, play a sound file.
subprocess.Popen(['start', 'alarm.wav'], shell=True)
```

After the while loop finishes, `alarm.wav` (or the sound file you choose) will play to notify the user that the countdown is over. On Windows, be sure to include 'start' in the list you pass to `Popen()` and pass the keyword argument `shell=True`. On OS X, pass 'open' instead of 'start' and remove `shell=True`.

Instead of playing a sound file, you could save a text file somewhere with a message like *Break time is over!* and use `Popen()` to open it at the end of the countdown. This will effectively create a pop-up window with a message. Or you could use the `webbrowser.open()` function to open a specific website at the end of the countdown. Unlike some free countdown application you'd find online, your own countdown program's alarm can be anything you want!

Ideas for Similar Programs

A countdown is a simple delay before continuing the program's execution. This can also be used for other applications and features, such as the following:

- Use `time.sleep()` to give the user a chance to press CTRL-C to cancel an action, such as deleting files. Your program can print a "Press CTRL-C to cancel" message and then handle any `KeyboardInterrupt` exceptions with `try` and `except` statements.
- For a long-term countdown, you can use `timedelta` objects to measure the number of days, hours, minutes, and seconds until some point (a birthday? an anniversary?) in the future.

Summary

The Unix epoch (January 1, 1970, at midnight, UTC) is a standard reference time for many programming languages, including Python. While the `time.time()` function module returns an epoch timestamp (that is, a float value of the number of seconds since the Unix epoch), the `datetime` module is better for performing date arithmetic and formatting or parsing strings with date information.

The `time.sleep()` function will block (that is, not return) for a certain number of seconds. It can be used to add pauses to your program. But if you want to schedule your programs to start at a certain time, the instructions at <http://nostarch.com/automatestuff/> can tell you how to use the scheduler already provided by your operating system.

The `threading` module is used to create multiple threads, which is useful when you need to download multiple files or do other tasks simultaneously. But make sure the thread reads and writes only local variables, or you might run into concurrency issues.

Finally, your Python programs can launch other applications with the `subprocess.Popen()` function. Command line arguments can be passed to the `Popen()` call to open specific documents with the application. Alternatively, you can use the `start`, `open`, or `see` program with `Popen()` to use your computer's file associations to automatically figure out which application to use to open a document. By using the other applications on your computer, your Python programs can leverage their capabilities for your automation needs.

Practice Questions

1. What is the Unix epoch?
2. What function returns the number of seconds since the Unix epoch?
3. How can you pause your program for exactly 5 seconds?
4. What does the `round()` function return?
5. What is the difference between a `datetime` object and a `timedelta` object?
6. Say you have a function named `spam()`. How can you call this function and run the code inside it in a separate thread?
7. What should you do to avoid concurrency issues with multiple threads?
8. How can you have your Python program run the `calc.exe` program located in the `C:\Windows\System32` folder?

Practice Projects

For practice, write programs that do the following.

Prettified Stopwatch

Expand the stopwatch project from this chapter so that it uses the `rjust()` and `ljust()` string methods to “prettify” the output. (These methods were covered in Chapter 6.) Instead of output such as this:

```
Lap #1: 3.56 (3.56)
Lap #2: 8.63 (5.07)
Lap #3: 17.68 (9.05)
Lap #4: 19.11 (1.43)
```

. . . the output will look like this:

```
Lap # 1: 3.56 ( 3.56)
Lap # 2: 8.63 ( 5.07)
Lap # 3: 17.68 ( 9.05)
Lap # 4: 19.11 ( 1.43)
```

Note that you will need string versions of the `lapNum`, `lapTime`, and `totalTime` integer and float variables in order to call the string methods on them.

Next, use the `pyperclip` module introduced in Chapter 6 to copy the text output to the clipboard so the user can quickly paste the output to a text file or email.

Scheduled Web Comic Downloader

Write a program that checks the websites of several web comics and automatically downloads the images if the comic was updated since the program’s last visit. Your operating system’s scheduler (Scheduled Tasks on Windows, launchd on OS X, and cron on Linux) can run your Python program once a day. The Python program itself can download the comic and then copy it to your desktop so that it is easy to find. This will free you from having to check the website yourself to see whether it has updated. (A list of web comics is available at <http://nostarch.com/automatestuff/>.)

16

SENDING EMAIL AND TEXT MESSAGES



Checking and replying to email is a huge time sink. Of course, you can't just write a program to handle all your email for you, since each message requires its own response.

But you can still automate plenty of email-related tasks once you know how to write programs that can send and receive email.

For example, maybe you have a spreadsheet full of customer records and want to send each customer a different form letter depending on their age and location details. Commercial software might not be able to do this for you; fortunately, you can write your own program to send these emails, saving yourself a lot of time copying and pasting form emails.

You can also write programs to send emails and SMS texts to notify you of things even while you're away from your computer. If you're automating a task that takes a couple of hours to do, you don't want to go back to your computer every few minutes to check on the program's status. Instead, the program can just text your phone when it's done—freeing you to focus on more important things while you're away from your computer.

SMTP

Much like HTTP is the protocol used by computers to send web pages across the Internet, *Simple Mail Transfer Protocol (SMTP)* is the protocol used for sending email. SMTP dictates how email messages should be formatted, encrypted, and relayed between mail servers, and all the other details that your computer handles after you click Send. You don't need to know these technical details, though, because Python's `smtplib` module simplifies them into a few functions.

SMTP just deals with sending emails to others. A different protocol, called IMAP, deals with retrieving emails sent to you and is described in “IMAP” on page 366.

Sending Email

You may be familiar with sending emails from Outlook or Thunderbird or through a website such as Gmail or Yahoo! Mail. Unfortunately, Python doesn't offer you a nice graphical user interface like those services. Instead, you call functions to perform each major step of SMTP, as shown in the following interactive shell example.

NOTE

Don't enter this example in IDLE; it won't work because `smtp.example.com`, `bob@example.com`, `MY_SECRET_PASSWORD`, and `alice@example.com` are just placeholders. This code is just an overview of the process of sending email with Python.

```
>>> import smtplib
>>> smtpObj = smtplib.SMTP('smtp.example.com', 587)
>>> smtpObj.ehlo()
(250, b'mx.example.com at your service, [216.172.148.131]\nSIZE 35882577\
n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nCHUNKING')
>>> smtpObj.starttls()
(220, b'2.0.0 Ready to start TLS')
>>> smtpObj.login('bob@example.com', 'MY_SECRET_PASSWORD')
(235, b'2.7.0 Accepted')
>>> smtpObj.sendmail('bob@example.com', 'alice@example.com', 'Subject: So
long.\nDear Alice, so long and thanks for all the fish. Sincerely, Bob')
{}
>>> smtpObj.quit()
(221, b'2.0.0 closing connection ko10sm23097611pb.52 - gsmtp')
```

In the following sections, we'll go through each step, replacing the placeholders with your information to connect and log in to an SMTP server, send an email, and disconnect from the server.

Connecting to an SMTP Server

If you've ever set up Thunderbird, Outlook, or another program to connect to your email account, you may be familiar with configuring the SMTP server and port. These settings will be different for each email provider, but a web search for *<your provider> smtp settings* should turn up the server and port to use.

The domain name for the SMTP server will usually be the name of your email provider's domain name, with *smtp*. in front of it. For example, Gmail's SMTP server is at *smtp.gmail.com*. Table 16-1 lists some common email providers and their SMTP servers. (The port is an integer value and will almost always be 587, which is used by the command encryption standard, TLS.)

Table 16-1: Email Providers and Their SMTP Servers

Provider	SMTP server domain name
Gmail	<i>smtp.gmail.com</i>
Outlook.com/Hotmail.com	<i>smtp-mail.outlook.com</i>
Yahoo Mail	<i>smtp.mail.yahoo.com</i>
AT&T	<i>smtp.mail.att.net</i> (port 465)
Comcast	<i>smtp.comcast.net</i>
Verizon	<i>smtp.verizon.net</i> (port 465)

Once you have the domain name and port information for your email provider, create an `SMTP` object by calling `smtplib.SMTP()`, passing the domain name as a string argument, and passing the port as an integer argument. The `SMTP` object represents a connection to an SMTP mail server and has methods for sending emails. For example, the following call creates an `SMTP` object for connecting to Gmail:

```
>>> smtpObj = smtplib.SMTP('smtp.gmail.com', 587)
>>> type(smtpObj)
<class 'smtplib.SMTP'>
```

Entering `type(smtpObj)` shows you that there's an `SMTP` object stored in `smtpObj`. You'll need this `SMTP` object in order to call the methods that log you in and send emails. If the `smtplib.SMTP()` call is not successful, your `SMTP` server might not support TLS on port 587. In this case, you will need to create an `SMTP` object using `smtplib.SMTP_SSL()` and port 465 instead.

```
>>> smtpObj = smtplib.SMTP_SSL('smtp.gmail.com', 465)
```

NOTE

If you are not connected to the Internet, Python will raise a `socket.gaierror`: [Errno 11004] getaddrinfo failed or similar exception.

For your programs, the differences between TLS and SSL aren't important. You only need to know which encryption standard your SMTP server uses so you know how to connect to it. In all of the interactive shell examples that follow, the `smtpObj` variable will contain an `SMTP` object returned by the `smtplib.SMTP()` or `smtplib.SMTP_SSL()` function.

Sending the SMTP “Hello” Message

Once you have the `SMTP` object, call its oddly named `ehlo()` method to “say hello” to the SMTP email server. This greeting is the first step in SMTP and is important for establishing a connection to the server. You don’t need to know the specifics of these protocols. Just be sure to call the `ehlo()` method first thing after getting the `SMTP` object or else the later method calls will result in errors. The following is an example of an `ehlo()` call and its return value:

```
>>> smtpObj.ehlo()
(250, b'mx.google.com at your service, [216.172.148.131]\nSIZE 35882577\
n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nCHUNKING')
```

If the first item in the returned tuple is the integer 250 (the code for “success” in SMTP), then the greeting succeeded.

Starting TLS Encryption

If you are connecting to port 587 on the SMTP server (that is, you’re using TLS encryption), you’ll need to call the `starttls()` method next. This required step enables encryption for your connection. If you are connecting to port 465 (using SSL), then encryption is already set up, and you should skip this step.

Here’s an example of the `starttls()` method call:

```
>>> smtpObj.starttls()
(220, b'2.0.0 Ready to start TLS')
```

`starttls()` puts your SMTP connection in TLS mode. The 220 in the return value tells you that the server is ready.

Logging in to the SMTP Server

Once your encrypted connection to the SMTP server is set up, you can log in with your username (usually your email address) and email password by calling the `login()` method.

```
>>> smtpObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')
(235, b'2.7.0 Accepted')
```

GMAIL'S APPLICATION-SPECIFIC PASSWORDS

Gmail has an additional security feature for Google accounts called *application-specific passwords*. If you receive an Application-specific password required error message when your program tries to log in, you will have to set up one of these passwords for your Python script. Check out the resources at <http://nostarch.com/automatestuff/> for detailed directions on how to set up an application-specific password for your Google account.

Pass a string of your email address as the first argument and a string of your password as the second argument. The 235 in the return value means authentication was successful. Python will raise an `smtplib.SMTPAuthenticationError` exception for incorrect passwords.

WARNING

Be careful about putting passwords in your source code. If anyone ever copies your program, they'll have access to your email account! It's a good idea to call `input()` and have the user type in the password. It may be inconvenient to have to enter a password each time you run your program, but this approach will prevent you from leaving your password in an unencrypted file on your computer where a hacker or laptop thief could easily get it.

Sending an Email

Once you are logged in to your email provider's SMTP server, you can call the `sendmail()` method to actually send the email. The `sendmail()` method call looks like this:

```
>>> smtpObj.sendmail('my_email_address@gmail.com', 'recipient@example.com',
'Subject: So long.\nDear Alice, so long and thanks for all the fish. Sincerely,
Bob')
{}
```

The `sendmail()` method requires three arguments.

- Your email address as a string (for the email's "from" address)
- The recipient's email address as a string or a list of strings for multiple recipients (for the "to" address)
- The email body as a string

The start of the email body string *must* begin with 'Subject: \n' for the subject line of the email. The '\n' newline character separates the subject line from the main body of the email.

The return value from `sendmail()` is a dictionary. There will be one key-value pair in the dictionary for each recipient for whom email delivery failed. An empty dictionary means all recipients were successfully sent the email.

Disconnecting from the SMTP Server

Be sure to call the `quit()` method when you are done sending emails. This will disconnect your program from the SMTP server.

```
>>> smtpObj.quit()
(221, b'2.0.0 closing connection ko10sm23097611pbd.52 - gsmtp')
```

The 221 in the return value means the session is ending.

To review all the steps for connecting and logging in to the server, sending email, and disconnection, see “[Sending Email](#)” on page 362.

IMAP

Just as SMTP is the protocol for sending email, the *Internet Message Access Protocol (IMAP)* specifies how to communicate with an email provider’s server to retrieve emails sent to your email address. Python comes with an `imaplib` module, but in fact the third-party `imapclient` module is easier to use. This chapter provides an introduction to using IMAPClient; the full documentation is at <http://imapclient.readthedocs.org/>.

The `imapclient` module downloads emails from an IMAP server in a rather complicated format. Most likely, you’ll want to convert them from this format into simple string values. The `pymail` module does the hard job of parsing these email messages for you. You can find the complete documentation for PyzMail at <http://www.magiksys.net/pyzmail/>.

Install `imapclient` and `pymail` from a Terminal window. Appendix A has steps on how to install third-party modules.

Retrieving and Deleting Emails with IMAP

Finding and retrieving an email in Python is a multistep process that requires both the `imapclient` and `pymail` third-party modules. Just to give you an overview, here’s a full example of logging in to an IMAP server, searching for emails, fetching them, and then extracting the text of the email messages from them.

```
>>> import imapclient
>>> imapObj = imapclient.IMAPClient('imap.gmail.com', ssl=True)
>>> imapObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')
'my_email_address@gmail.com Jane Doe authenticated (Success)'
>>> imapObj.select_folder('INBOX', readonly=True)
>>> UIDs = imapObj.search(['SINCE 05-Jul-2014'])
>>> UIDs
[40032, 40033, 40034, 40035, 40036, 40037, 40038, 40039, 40040, 40041]
>>> rawMessages = imapObj.fetch([40041], ['BODY[]', 'FLAGS'])
>>> import pymail
>>> message = pymail.PyzMessage.factory(rawMessages[40041]['BODY[]'])
>>> message.get_subject()
'Hello!'
```

```
>>> message.get_addresses('from')
[('Edward Snowden', 'esnowden@nsa.gov')]
>>> message.get_addresses('to')
[('Jane Doe', 'jdoe@example.com')]
>>> message.get_addresses('cc')
[]
>>> message.get_addresses('bcc')
[]
>>> message.text_part != None
True
>>> message.text_part.get_payload().decode(message.text_part.charset)
'Follow the money.\r\n\r\n-Ed\r\n'
>>> message.html_part != None
True
>>> message.html_part.get_payload().decode(message.html_part.charset)
'<div dir="ltr"><div>So long, and thanks for all the fish!<br><br></div>-Al<br></div>\r\n'
>>> imapObj.logout()
```

You don't have to memorize these steps. After we go through each step in detail, you can come back to this overview to refresh your memory.

Connecting to an IMAP Server

Just like you needed an SMTP object to connect to an SMTP server and send email, you need an `IMAPClient` object to connect to an IMAP server and receive email. First you'll need the domain name of your email provider's IMAP server. This will be different from the SMTP server's domain name. Table 16-2 lists the IMAP servers for several popular email providers.

Table 16-2: Email Providers and Their IMAP Servers

Provider	IMAP server domain name
Gmail	<code>imap.gmail.com</code>
Outlook.com/Hotmail.com	<code>imap-mail.outlook.com</code>
Yahoo Mail	<code>imap.mail.yahoo.com</code>
AT&T	<code>imap.mail.att.net</code>
Comcast	<code>imap.comcast.net</code>
Verizon	<code>incoming.verizon.net</code>

Once you have the domain name of the IMAP server, call the `imapclient.IMAPClient()` function to create an `IMAPClient` object. Most email providers require SSL encryption, so pass the `ssl=True` keyword argument. Enter the following into the interactive shell (using your provider's domain name):

```
>>> import imapclient
>>> imapObj = imapclient.IMAPClient('imap.gmail.com', ssl=True)
```

In all of the interactive shell examples in the following sections, the `imapObj` variable will contain an `IMAPClient` object returned from the `imapclient.IMAPClient()` function. In this context, a *client* is the object that connects to the server.

Logging in to the IMAP Server

Once you have an `IMAPClient` object, call its `login()` method, passing in the username (this is usually your email address) and password as strings.

```
>>> imapObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')
'my_email_address@gmail.com Jane Doe authenticated (Success)'
```

WARNING

Remember, never write a password directly into your code! Instead, design your program to accept the password returned from `input()`.

If the IMAP server rejects this username/password combination, Python will raise an `imaplib.error` exception. For Gmail accounts, you may need to use an application-specific password; for more details, see “Gmail’s Application-Specific Passwords” on page 365.

Searching for Email

Once you’re logged on, actually retrieving an email that you’re interested in is a two-step process. First, you must select a folder you want to search through. Then, you must call the `IMAPClient` object’s `search()` method, passing in a string of IMAP search keywords.

Selecting a Folder

Almost every account has an `INBOX` folder by default, but you can also get a list of folders by calling the `IMAPClient` object’s `list_folders()` method. This returns a list of tuples. Each tuple contains information about a single folder. Continue the interactive shell example by entering the following:

```
>>> import pprint
>>> pprint.pprint(imapObj.list_folders())
[('\'\HasNoChildren',), '/', 'Drafts'),
 ('\'\HasNoChildren',), '/', 'Filler'),
 ('\'\HasNoChildren',), '/', 'INBOX'),
 ('\'\HasNoChildren',), '/', 'Sent'),
 --snip--
 ('\'\HasNoChildren', '\Flagged'), '/', '[Gmail]/Starred'),
 ('\'\HasNoChildren', '\Trash'), '/', '[Gmail]/Trash')]
```

This is what your output might look like if you have a Gmail account. (Gmail calls its folders *labels*, but they work the same way as folders.) The three values in each of the tuples—for example, (('\HasNoChildren',), '/', 'INBOX')—are as follows:

- A tuple of the folder's flags. (Exactly what these flags represent is beyond the scope of this book, and you can safely ignore this field.)
- The delimiter used in the name string to separate parent folders and subfolders.
- The full name of the folder.

To select a folder to search through, pass the folder's name as a string into the `IMAPClient` object's `select_folder()` method.

```
>>> imapObj.select_folder('INBOX', readonly=True)
```

You can ignore `select_folder()`'s return value. If the selected folder does not exist, Python will raise an `imaplib.error` exception.

The `readonly=True` keyword argument prevents you from accidentally making changes or deletions to any of the emails in this folder during the subsequent method calls. Unless you *want* to delete emails, it's a good idea to always set `readonly` to True.

Performing the Search

With a folder selected, you can now search for emails with the `IMAPClient` object's `search()` method. The argument to `search()` is a list of strings, each formatted to the IMAP's search keys. Table 16-3 describes the various search keys.

Table 16-3: IMAP Search Keys

Search key	Meaning
'ALL'	Returns all messages in the folder. You may run into <code>imaplib</code> size limits if you request all the messages in a large folder. See "Size Limits" on page 371.
'BEFORE date', 'ON date', 'SINCE date'	These three search keys return, respectively, messages that were received by the IMAP server before, on, or after the given <i>date</i> . The date must be formatted like 05-Jul-2015. Also, while 'SINCE 05-Jul-2015' will match messages on and after July 5, 'BEFORE 05-Jul-2015' will match only messages before July 5 but not on July 5 itself.
'SUBJECT string', 'BODY string', 'TEXT string'	Returns messages where <i>string</i> is found in the subject, body, or either, respectively. If <i>string</i> has spaces in it, then enclose it with double quotes: 'TEXT "search with spaces"'.

(continued)

Table 16-3 (continued)

Search key	Meaning
'FROM <i>string'</i> , 'TO <i>string'</i> , 'CC <i>string'</i> , 'BCC <i>string'</i>	Returns all messages where <i>string</i> is found in the "from" emailaddress, "to" addresses, "cc" (carbon copy) addresses, or "bcc" (blind carbon copy) addresses, respectively. If there are multiple email addresses in <i>string</i> , then separate them with spaces and enclose them all with double quotes: 'CC "firstcc@example.com secondcc@example.com"'.
'SEEN', 'UNSEEN'	Returns all messages with and without the \Seen flag, respectively. An email obtains the \Seen flag if it has been accessed with a <code>fetch()</code> method call (described later) or if it is clicked when you're checking your email in an email program or web browser. It's more common to say the email has been "read" rather than "seen," but they mean the same thing.
'ANSWERED', 'UNANSWERED'	Returns all messages with and without the \Answered flag, respectively. A message obtains the \Answered flag when it is replied to.
'DELETED', 'UNDELETED'	Returns all messages with and without the \Deleted flag, respectively. Email messages deleted with the <code>delete_messages()</code> method are given the \Deleted flag but are not permanently deleted until the <code>expunge()</code> method is called (see "Deleting Emails" on page 375). Note that some email providers, such as Gmail, automatically expunge emails.
'DRAFT', 'UNDRAFT'	Returns all messages with and without the \Draft flag, respectively. Draft messages are usually kept in a separate Drafts folder rather than in the INBOX folder.
'FLAGGED', 'UNFLAGGED'	Returns all messages with and without the \Flagged flag, respectively. This flag is usually used to mark email messages as "Important" or "Urgent."
'LARGER <i>N'</i> , 'SMALLER <i>N'</i>	Returns all messages larger or smaller than <i>N</i> bytes, respectively.
'NOT <i>search-key'</i>	Returns the messages that <i>search-key</i> would <i>not</i> have returned.
'OR <i>search-key1</i> <i>search-key2'</i>	Returns the messages that match <i>either</i> the first or second <i>search-key</i> .

Note that some IMAP servers may have slightly different implementations for how they handle their flags and search keys. It may require some experimentation in the interactive shell to see exactly how they behave.

You can pass multiple IMAP search key strings in the list argument to the `search()` method. The messages returned are the ones that match *all* the search keys. If you want to match *any* of the search keys, use the OR search key. For the NOT and OR search keys, one and two complete search keys follow the NOT and OR, respectively.

Here are some example `search()` method calls along with their meanings:

`imapObj.search(['ALL'])` Returns every message in the currently selected folder.

`imapObj.search(['ON 05-Jul-2015'])` Returns every message sent on July 5, 2015.

```
imapObj.search(['SINCE 01-Jan-2015', 'BEFORE 01-Feb-2015', 'UNSEEN'])
```

Returns every message sent in January 2015 that is unread. (Note that this means *on and after* January 1 and up to *but not including* February 1.)

```
imapObj.search(['SINCE 01-Jan-2015', 'FROM alice@example.com'])
```

Returns every message from *alice@example.com* sent since the start of 2015.

```
imapObj.search(['SINCE 01-Jan-2015', 'NOT FROM alice@example.com'])
```

Returns every message sent from everyone except *alice@example.com* since the start of 2015.

```
imapObj.search(['OR FROM alice@example.com FROM bob@example.com'])
```

Returns every message ever sent from *alice@example.com* or *bob@example.com*.

```
imapObj.search(['FROM alice@example.com', 'FROM bob@example.com'])
```

Trick example! This search will never return any messages, because messages must match *all* search keywords. Since there can be only one “from” address, it is impossible for a message to be from both *alice@example.com* and *bob@example.com*.

The `search()` method doesn’t return the emails themselves but rather unique IDs (UIDs) for the emails, as integer values. You can then pass these UIDs to the `fetch()` method to obtain the email content.

Continue the interactive shell example by entering the following:

```
>>> UIDs = imapObj.search(['SINCE 05-Jul-2015'])
>>> UIDs
[40032, 40033, 40034, 40035, 40036, 40037, 40038, 40039, 40040, 40041]
```

Here, the list of message IDs (for messages received July 5 onward) returned by `search()` is stored in `UIDs`. The list of UIDs returned on your computer will be different from the ones shown here; they are unique to a particular email account. When you later pass UIDs to other function calls, use the UID values you received, not the ones printed in this book’s examples.

Size Limits

If your search matches a large number of email messages, Python might raise an exception that says `imaplib.error: got more than 10000 bytes`. When this happens, you will have to disconnect and reconnect to the IMAP server and try again.

This limit is in place to prevent your Python programs from eating up too much memory. Unfortunately, the default size limit is often too small. You can change this limit from 10,000 bytes to 10,000,000 bytes by running this code:

```
>>> import imaplib
>>> imaplib._MAXLINE = 10000000
```

This should prevent this error message from coming up again. You may want to make these two lines part of every IMAP program you write.

USING IMAPCLIENT'S GMAIL_SEARCH() METHOD

If you are logging in to the `imap.gmail.com` server to access a Gmail account, the `IMAPClient` object provides an extra search function that mimics the search bar at the top of the Gmail web page, as highlighted in Figure 16-1.



Figure 16-1: The search bar at the top of the Gmail web page

Instead of searching with IMAP search keys, you can use Gmail's more sophisticated search engine. Gmail does a good job of matching closely related words (for example, a search for `driving` will also match `drive` and `drove`) and sorting the search results by most significant matches. You can also use Gmail's advanced search operators (see <http://nostarch.com/automatestuff/> for more information). If you are logging in to a Gmail account, pass the search terms to the `gmail_search()` method instead of the `search()` method, like in the following interactive shell example:

```
>>> UIDs = imapObj.gmail_search('meaning of life')
>>> UIDs
[42]
```

Ah, yes—that's that email with the meaning of life! I was looking for that.

Fetching an Email and Marking It As Read

Once you have a list of UIDs, you can call the `IMAPClient` object's `fetch()` method to get the actual email content.

The list of UIDs will be `fetch()`'s first argument. The second argument should be the list `['BODY[]']`, which tells `fetch()` to download all the body content for the emails specified in your UID list.

Let's continue our interactive shell example.

```
>>> rawMessages = imapObj.fetch(UIDs, ['BODY[]'])
>>> import pprint
>>> pprint.pprint(rawMessages)
{40040: {'BODY[]': 'Delivered-To: my_email_address@gmail.com\r\n'
                  'Received: by 10.76.71.167 with SMTP id '
--snip--
                  '\r\n'
                  '-----_Part_6000970_707736290.1404819487066--\r\n',
 'SEQ': 5430}}
```

Import `pprint` and pass the return value from `fetch()`, stored in the variable `rawMessages`, to `pprint.pprint()` to “pretty print” it, and you’ll see that this return value is a nested dictionary of messages with UIDs as the keys. Each message is stored as a dictionary with two keys: `'BODY[]'` and `'SEQ'`. The `'BODY[]'` key maps to the actual body of the email. The `'SEQ'` key is for a *sequence number*, which has a similar role to the UID. You can safely ignore it.

As you can see, the message content in the `'BODY[]'` key is pretty unintelligible. It’s in a format called RFC 822, which is designed for IMAP servers to read. But you don’t need to understand the RFC 822 format; later in this chapter, the `pymail` module will make sense of it for you.

When you selected a folder to search through, you called `select_folder()` with the `readonly=True` keyword argument. Doing this will prevent you from accidentally deleting an email—but it also means that emails will not get marked as read if you fetch them with the `fetch()` method. If you *do* want emails to be marked as read when you fetch them, you will need to pass `readonly=False` to `select_folder()`. If the selected folder is already in readonly mode, you can reselect the current folder with another call to `select_folder()`, this time with the `readonly=False` keyword argument:

```
>>> imapObj.select_folder('INBOX', readonly=False)
```

Getting Email Addresses from a Raw Message

The raw messages returned from the `fetch()` method still aren’t very useful to people who just want to read their email. The `pymail` module parses these raw messages and returns them as `PyzMessage` objects, which make the subject, body, “To” field, “From” field, and other sections of the email easily accessible to your Python code.

Continue the interactive shell example with the following (using UIDs from your own email account, not the ones shown here):

```
>>> import pymail
>>> message = pymail.PyzMessage.factory(rawMessages[40041]['BODY[]'])
```

First, import `pymail`. Then, to create a `PyzMessage` object of an email, call the `pymail.PyzMessage.factory()` function and pass it the `'BODY[]'` section of the raw message. Store the result in `message`. Now `message` contains

a PyzMessage object, which has several methods that make it easy to get the email's subject line, as well as all sender and recipient addresses. The `get_subject()` method returns the subject as a simple string value. The `get_addresses()` method returns a list of addresses for the field you pass it. For example, the method calls might look like this:

```
>>> message.get_subject()
'Hello!'
>>> message.get_addresses('from')
[('Edward Snowden', 'esnowden@nsa.gov')]
>>> message.get_addresses('to')
[(Jane Doe', 'my_email_address@gmail.com')]
>>> message.get_addresses('cc')
[]
>>> message.get_addresses('bcc')
[]
```

Notice that the argument for `get_addresses()` is '`from`', '`to`', '`cc`', or '`bcc`'. The return value of `get_addresses()` is a list of tuples. Each tuple contains two strings: The first is the name associated with the email address, and the second is the email address itself. If there are no addresses in the requested field, `get_addresses()` returns a blank list. Here, the '`cc`' carbon copy and '`bcc`' blind carbon copy fields both contained no addresses and so returned empty lists.

Getting the Body from a Raw Message

Emails can be sent as plaintext, HTML, or both. Plaintext emails contain only text, while HTML emails can have colors, fonts, images, and other features that make the email message look like a small web page. If an email is only plaintext, its PyzMessage object will have its `html_part` attribute set to `None`. Likewise, if an email is only HTML, its PyzMessage object will have its `text_part` attribute set to `None`.

Otherwise, the `text_part` or `html_part` value will have a `get_payload()` method that returns the email's body as a value of the `bytes` data type. (The `bytes` data type is beyond the scope of this book.) But this *still* isn't a string value that we can use. Ugh! The last step is to call the `decode()` method on the `bytes` value returned by `get_payload()`. The `decode()` method takes one argument: the message's character encoding, stored in the `text_part.charset` or `html_part.charset` attribute. This, finally, will return the string of the email's body.

Continue the interactive shell example by entering the following:

```
❶ >>> message.text_part != None
True
>>> message.text_part.get_payload().decode(message.text_part.charset)
❷ 'So long, and thanks for all the fish!\r\n\r\nA1\r\n'
❸ >>> message.html_part != None
True
```

```
❸ >>> message.html_part.get_payload().decode(message.html_part.charset)
'<div dir="ltr"><div>So long, and thanks for all the fish!<br><br></div>-Al
<br></div>\r\n'
```

The email we’re working with has both plaintext and HTML content, so the `PyzMessage` object stored in `message` has `text_part` and `html_part` attributes not equal to `None` ❶ ❷. Calling `get_payload()` on the message’s `text_part` and then calling `decode()` on the bytes value returns a string of the text version of the email ❸. Using `get_payload()` and `decode()` with the message’s `html_part` returns a string of the HTML version of the email ❹.

Deleting Emails

To delete emails, pass a list of message UIDs to the `IMAPClient` object’s `delete_messages()` method. This marks the emails with the `\Deleted` flag. Calling the `expunge()` method will permanently delete all emails with the `\Deleted` flag in the currently selected folder. Consider the following interactive shell example:

```
❶ >>> imapObj.select_folder('INBOX', readonly=False)
❷ >>> UIDs = imapObj.search(['ON 09-Jul-2015'])
>>> UIDs
[40066]
>>> imapObj.delete_messages(UIDs)
❸ {40066: ('\\\Seen', '\\Deleted')}
>>> imapObj.expunge()
('Success', [(5452, 'EXISTS')])
```

Here we select the inbox by calling `select_folder()` on the `IMAPClient` object and passing ‘INBOX’ as the first argument; we also pass the keyword argument `readonly=False` so that we can delete emails ❶. We search the inbox for messages received on a specific date and store the returned message IDs in `UIDs` ❷. Calling `delete_message()` and passing it `UIDs` returns a dictionary; each key-value pair is a message ID and a tuple of the message’s flags, which should now include `\Deleted` ❸. Calling `expunge()` then permanently deletes messages with the `\Deleted` flag and returns a success message if there were no problems expunging the emails. Note that some email providers, such as Gmail, automatically expunge emails deleted with `delete_messages()` instead of waiting for an `expunge` command from the IMAP client.

Disconnecting from the IMAP Server

When your program has finished retrieving or deleting emails, simply call the `IMAPClient`’s `logout()` method to disconnect from the IMAP server.

```
>>> imapObj.logout()
```

If your program runs for several minutes or more, the IMAP server may *time out*, or automatically disconnect. In this case, the next method call your program makes on the `IMAPClient` object will raise an exception like the following:

```
imaplib.abort: socket error: [WinError 10054] An existing connection was  
forcibly closed by the remote host
```

In this event, your program will have to call `imapclient.IMAPClient()` to connect again.

Whew! That's it. There were a lot of hoops to jump through, but you now have a way to get your Python programs to log in to an email account and fetch emails. You can always consult the overview in "Retrieving and Deleting Emails with IMAP" on page 366 whenever you need to remember all of the steps.

Project: Sending Member Dues Reminder Emails

Say you have been "volunteered" to track member dues for the Mandatory Volunteerism Club. This is a truly boring job, involving maintaining a spreadsheet of everyone who has paid each month and emailing reminders to those who haven't. Instead of going through the spreadsheet yourself and copying and pasting the same email to everyone who is behind on dues, let's—you guessed it—write a script that does this for you.

At a high level, here's what your program will do:

- Read data from an Excel spreadsheet.
- Find all members who have not paid dues for the latest month.
- Find their email addresses and send them personalized reminders.

This means your code will need to do the following:

- Open and read the cells of an Excel document with the `openpyxl` module. (See Chapter 12 for working with Excel files.)
- Create a dictionary of members who are behind on their dues.
- Log in to an SMTP server by calling `smtplib.SMTP()`, `ehlo()`, `starttls()`, and `login()`.
- For all members behind on their dues, send a personalized reminder email by calling the `sendmail()` method.

Open a new file editor window and save it as `sendDuesReminders.py`.

Step 1: Open the Excel File

Let's say the Excel spreadsheet you use to track membership dues payments looks like Figure 16-2 and is in a file named `duesRecords.xlsx`. You can download this file from <http://nostarch.com/automatestuff/>.

	A	B	C	D	E	F	G	H
1	Member	Email	Jan 2014	Feb 2014	Mar 2014	Apr 2014	May 2014	Jun 2014
2	Alice	alice@example.com	paid	paid	paid	paid	paid	
3	Bob	bob@example.com	paid	paid	paid	paid		
4	Carol	carol@example.com	paid	paid	paid	paid	paid	
5	David	david@example.com	paid	paid	paid	paid	paid	paid
6	Eve	eve@example.com	paid	paid	paid			
7	Fred	fred@example.com	paid	paid	paid	paid	paid	paid
8								
9								

Figure 16-2: The spreadsheet for tracking member dues payments

This spreadsheet has every member’s name and email address. Each month has a column tracking members’ payment statuses. The cell for each member is marked with the text *paid* once they have paid their dues.

The program will have to open *duesRecords.xlsx* and figure out the column for the latest month by calling the `get_highest_column()` method. (You can consult Chapter 12 for more information on accessing cells in Excel spreadsheet files with the `openpyxl` module.) Enter the following code into the file editor window:

```

#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsheet.

import openpyxl, smtplib, sys

# Open the spreadsheet and get the latest dues status.
❶ wb = openpyxl.load_workbook('duesRecords.xlsx')
❷ sheet = wb.get_sheet_by_name('Sheet1')

❸ lastCol = sheet.get_highest_column()
❹ latestMonth = sheet.cell(row=1, column=lastCol).value

# TODO: Check each member's payment status.

# TODO: Log in to email account.

# TODO: Send out reminder emails.

```

After importing the `openpyxl`, `smtplib`, and `sys` modules, we open our *duesRecords.xlsx* file and store the resulting `Workbook` object in `wb` ❶. Then we get Sheet 1 and store the resulting `Worksheet` object in `sheet` ❷. Now that we have a `Worksheet` object, we can access rows, columns, and cells. We store the highest column in `lastCol` ❸, and we then use row number 1 and `lastCol` to access the cell that should hold the most recent month. We get the value in this cell and store it in `latestMonth` ❹.

Step 2: Find All Unpaid Members

Once you've determined the column number of the latest month (stored in `lastCol`), you can loop through all rows after the first row (which has the column headers) to see which members have the text *paid* in the cell for that month's dues. If the member hasn't paid, you can grab the member's name and email address from columns 1 and 2, respectively. This information will go into the `unpaidMembers` dictionary, which will track all members who haven't paid in the most recent month. Add the following code to `sendDuesReminder.py`.

```
#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsheet.

--snip--

# Check each member's payment status.
unpaidMembers = {}
❶ for r in range(2, sheet.get_highest_row() + 1):
❷     payment = sheet.cell(row=r, column=lastCol).value
        if payment != 'paid':
❸         name = sheet.cell(row=r, column=1).value
❹         email = sheet.cell(row=r, column=2).value
❺         unpaidMembers[name] = email
```

This code sets up an empty dictionary `unpaidMembers` and then loops through all the rows after the first ❶. For each row, the value in the most recent column is stored in `payment` ❷. If `payment` is not equal to 'paid', then the value of the first column is stored in `name` ❸, the value of the second column is stored in `email` ❹, and `name` and `email` are added to `unpaidMembers` ❺.

Step 3: Send Customized Email Reminders

Once you have a list of all unpaid members, it's time to send them email reminders. Add the following code to your program, except with your real email address and provider information:

```
#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsheet.

--snip--

# Log in to email account.
smtpObj = smtplib.SMTP('smtp.gmail.com', 587)
smtpObj.ehlo()
smtpObj.starttls()
smtpObj.login('my_email_address@gmail.com', sys.argv[1])
```

Create an SMTP object by calling `smtplib.SMTP()` and passing it the domain name and port for your provider. Call `ehlo()` and `starttls()`, and then call `login()` and pass it your email address and `sys.argv[1]`, which will store your password string. You'll enter the password as a command line argument each time you run the program, to avoid saving your password in your source code.

Once your program has logged in to your email account, it should go through the `unpaidMembers` dictionary and send a personalized email to each member's email address. Add the following to `sendDuesReminders.py`:

```
#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsheet.

--snip--

# Send out reminder emails.
for name, email in unpaidMembers.items():
   ❶    body = "Subject: %s dues unpaid.\nDear %s,\nRecords show that you have not
            paid dues for %. Please make this payment as soon as possible. Thank you!" % 
            (latestMonth, name, latestMonth)
   ❷    print('Sending email to %s...' % email)
   ❸    sendmailStatus = smtpObj.sendmail('my_email_address@gmail.com', email, body)

   ❹    if sendmailStatus != {}:
        print('There was a problem sending email to %s: %s' % (email,
            sendmailStatus))
smtpObj.quit()
```

This code loops through the names and emails in `unpaidMembers`. For each member who hasn't paid, we customize a message with the latest month and the member's name, and store the message in `body` ❶. We print output saying that we're sending an email to this member's email address ❷. Then we call `sendmail()`, passing it the from address and the customized message ❸. We store the return value in `sendmailStatus`.

Remember that the `sendmail()` method will return a nonempty dictionary value if the SMTP server reported an error sending that particular email. The last part of the for loop at ❹ checks if the returned dictionary is nonempty, and if it is, prints the recipient's email address and the returned dictionary.

After the program is done sending all the emails, the `quit()` method is called to disconnect from the SMTP server.

When you run the program, the output will look something like this:

```
Sending email to alice@example.com...
Sending email to bob@example.com...
Sending email to eve@example.com...
```

The recipients will receive an email that looks like Figure 16-3.



Figure 16-3: An automatically sent email from `sendDuesReminders.py`

Sending Text Messages with Twilio

Most people are more likely to be near their phones than their computers, so text messages can be a more immediate and reliable way of sending notifications than email. Also, the short length of text messages makes it more likely that a person will get around to reading them.

In this section, you'll learn how to sign up for the free Twilio service and use its Python module to send text messages. Twilio is an *SMS gateway service*, which means it's a service that allows you to send text messages from your programs. Although you will be limited in how many texts you can send per month and the texts will be prefixed with the words *Sent from a Twilio trial account*, this trial service is probably adequate for your personal programs. The free trial is indefinite; you won't have to upgrade to a paid plan later.

Twilio isn't the only SMS gateway service. If you prefer not to use Twilio, you can find alternative services by searching online for *free sms gateway*, *python sms api*, or even *twilio alternatives*.

Before signing up for a Twilio account, install the `twilio` module. Appendix A has more details about installing third-party modules.

NOTE

This section is specific to the United States. Twilio does offer SMS texting services for countries outside of the United States, but those specifics aren't covered in this book. The `twilio` module and its functions, however, will work the same outside the United States. See <http://twilio.com/> for more information.

Signing Up for a Twilio Account

Go to <http://twilio.com/> and fill out the sign-up form. Once you've signed up for a new account, you'll need to verify a mobile phone number that you want to send texts to. (This verification is necessary to prevent people from using the service to spam random phone numbers with text messages.)

After receiving the text with the verification number, enter it into the Twilio website to prove that you own the mobile phone you are verifying. You will now be able to send texts to this phone number using the `twilio` module.

Twilio provides your trial account with a phone number to use as the sender of text messages. You will need two more pieces of information: your account SID and the auth (authentication) token. You can find this information on the Dashboard page when you are logged in to your Twilio account. These values act as your Twilio username and password when logging in from a Python program.

Sending Text Messages

Once you've installed the `twilio` module, signed up for a Twilio account, verified your phone number, registered a Twilio phone number, and obtained your account SID and auth token, you will finally be ready to send yourself text messages from your Python scripts.

Compared to all the registration steps, the actual Python code is fairly simple. With your computer connected to the Internet, enter the following into the interactive shell, replacing the `accountSID`, `authToken`, `myTwilioNumber`, and `myCellPhone` variable values with your real information:

```
❶ >>> from twilio.rest import TwilioRestClient
>>> accountSID = 'ACxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
>>> authToken = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
❷ >>> twilioCli = TwilioRestClient(accountSID, authToken)
>>> myTwilioNumber = '+14955551234'
>>> myCellPhone = '+14955558888'
❸ >>> message = twilioCli.messages.create(body='Mr. Watson - Come here - I want
to see you.', from_=myTwilioNumber, to=myCellPhone)
```

A few moments after typing the last line, you should receive a text message that reads *Sent from your Twilio trial account - Mr. Watson - Come here - I want to see you.*

Because of the way the `twilio` module is set up, you need to import it using `from twilio.rest import TwilioRestClient`, not just `import twilio` ❶. Store your account SID in `accountSID` and your auth token in `authToken` and then call `TwilioRestClient()` and pass it `accountSID` and `authToken`. The call to `TwilioRestClient()` returns a `TwilioRestClient` object ❷. This object has a `messages` attribute, which in turn has a `create()` method you can use to send text messages. This is the method that will instruct Twilio's servers to send your text message. After storing your Twilio number and cell phone number in `myTwilioNumber` and `myCellPhone`, respectively, call `create()` and pass it keyword arguments specifying the body of the text message, the sender's number (`myTwilioNumber`), and the recipient's number (`myCellPhone`) ❸.

The `Message` object returned from the `create()` method will have information about the text message that was sent. Continue the interactive shell example by entering the following:

```
>>> message.to
'+14955558888'
>>> message.from_
'+14955551234'
>>> message.body
'Mr. Watson - Come here - I want to see you.'
```

The `to`, `from_`, and `body` attributes should hold your cell phone number, Twilio number, and message, respectively. Note that the sending phone number is in the `from_` attribute—with an underscore at the end—not `from`. This is because `from` is a keyword in Python (you've seen it used in the `from`

`modulename import *` form of import statement, for example), so it cannot be used as an attribute name. Continue the interactive shell example with the following:

```
>>> message.status  
'queued'  
>>> message.date_created  
datetime.datetime(2015, 7, 8, 1, 36, 18)  
>>> message.date_sent == None  
True
```

The `status` attribute should give you a string. The `date_created` and `date_sent` attributes should give you a `datetime` object if the message has been created and sent. It may seem odd that the `status` attribute is set to `'queued'` and the `date_sent` attribute is set to `None` when you've already received the text message. This is because you captured the `Message` object in the `message` variable *before* the text was actually sent. You will need to refetch the `Message` object in order to see its most up-to-date `status` and `date_sent`. Every Twilio message has a unique string ID (SID) that can be used to fetch the latest update of the `Message` object. Continue the interactive shell example by entering the following:

```
>>> message.sid  
'SM09520de7639ba3af137c6fcb7c5f4b51'  
❶ >>> updatedMessage = twilioCli.messages.get(message.sid)  
>>> updatedMessage.status  
'delivered'  
>>> updatedMessage.date_sent  
datetime.datetime(2015, 7, 8, 1, 36, 18)
```

Entering `message.sid` show you this message's long SID. By passing this SID to the Twilio client's `get()` method ❶, you can retrieve a new `Message` object with the most up-to-date information. In this new `Message` object, the `status` and `date_sent` attributes are correct.

The `status` attribute will be set to one of the following string values: `'queued'`, `'sending'`, `'sent'`, `'delivered'`, `'undelivered'`, or `'failed'`. These statuses are self-explanatory, but for more precise details, take a look at the resources at <http://nostarch.com/automatestuff/>.

RECEIVING TEXT MESSAGES WITH PYTHON

Unfortunately, receiving text messages with Twilio is a bit more complicated than sending them. Twilio requires that you have a website running its own web application. That's beyond the scope of this book, but you can find more details in the resources for this book (<http://nostarch.com/automatestuff/>).

Project: “Just Text Me” Module

The person you’ll most often text from your programs is probably you. Texting is a great way to send yourself notifications when you’re away from your computer. If you’ve automated a boring task with a program that takes a couple of hours to run, you could have it notify you with a text when it’s finished. Or you may have a regularly scheduled program running that sometimes needs to contact you, such as a weather-checking program that texts you a reminder to pack an umbrella.

As a simple example, here’s a small Python program with a `textmyself()` function that sends a message passed to it as a string argument. Open a new file editor window and enter the following code, replacing the account SID, auth token, and phone numbers with your own information. Save it as `textMyself.py`.

```
#! python3
# textMyself.py - Defines the textmyself() function that texts a message
# passed to it as a string.

# Preset values:
accountSID = 'ACxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
authToken = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
myNumber = '+15559998888'
twilioNumber = '+15552225678'

from twilio.rest import TwilioRestClient

❶ def textmyself(message):
❷     twilioCli = TwilioRestClient(accountSID, authToken)
❸     twilioCli.messages.create(body=message, from_=twilioNumber, to=myNumber)
```

This program stores an account SID, auth token, sending number, and receiving number. It then defined `textmyself()` to take on argument ❶, make a `TwilioRestClient` object ❷, and call `create()` with the message you passed ❸.

If you want to make the `textmyself()` function available to your other programs, simply place the `textMyself.py` file in the same folder as the Python executable (`C:\Python34` on Windows, `/usr/local/lib/python3.4` on OS X, and `/usr/bin/python3` on Linux). Now you can use the function in your other programs. Whenever you want one of your programs to text you, just add the following:

```
import textmyself
textmyself.textmyself('The boring task is finished.')
```

You need to sign up for Twilio and write the texting code only once. After that, it’s just two lines of code to send a text from any of your other programs.

Summary

We communicate with each other on the Internet and over cell phone networks in dozens of different ways, but email and texting predominate. Your programs can communicate through these channels, which gives them powerful new notification features. You can even write programs running on different computers that communicate with one another directly via email, with one program sending emails with SMTP and the other retrieving them with IMAP.

Python's `smtplib` provides functions for using the SMTP to send emails through your email provider's SMTP server. Likewise, the third-party `imapclient` and `pymail` modules let you access IMAP servers and retrieve emails sent to you. Although IMAP is a bit more involved than SMTP, it's also quite powerful and allows you to search for particular emails, download them, and parse them to extract the subject and body as string values.

Texting is a bit different from email, since, unlike email, more than just an Internet connection is needed to send SMS texts. Fortunately, services such as Twilio provide modules to allow you to send text messages from your programs. Once you go through an initial setup process, you'll be able to send texts with just a couple lines of code.

With these modules in your skill set, you'll be able to program the specific conditions under which your programs should send notifications or reminders. Now your programs will have reach far beyond the computer they're running on!

Practice Questions

1. What is the protocol for sending email? For checking and receiving email?
2. What four `smtplib` functions/methods must you call to log in to an SMTP server?
3. What two `imapclient` functions/methods must you call to log in to an IMAP server?
4. What kind of argument do you pass to `imapObj.search()`?
5. What do you do if your code gets an error message that says got `more than 10000 bytes`?
6. The `imapclient` module handles connecting to an IMAP server and finding emails. What is one module that handles reading the emails that `imapclient` collects?
7. What three pieces of information do you need from Twilio before you can send text messages?

Practice Projects

For practice, write programs that do the following.

Random Chore Assignment Emailer

Write a program that takes a list of people's email addresses and a list of chores that need to be done and randomly assigns chores to people. Email each person their assigned chores. If you're feeling ambitious, keep a record of each person's previously assigned chores so that you can make sure the program avoids assigning anyone the same chore they did last time. For another possible feature, schedule the program to run once a week automatically.

Here's a hint: If you pass a list to the `random.choice()` function, it will return a randomly selected item from the list. Part of your code could look like this:

```
chores = ['dishes', 'bathroom', 'vacuum', 'walk dog']
randomChore = random.choice(chores)
chores.remove(randomChore)    # this chore is now taken, so remove it
```

Umbrella Reminder

Chapter 11 showed you how to use the `requests` module to scrape data from <http://weather.gov/>. Write a program that runs just before you wake up in the morning and checks whether it's raining that day. If so, have the program text you a reminder to pack an umbrella before leaving the house.

Auto Unsubscriber

Write a program that scans through your email account, finds all the unsubscribe links in all your emails, and automatically opens them in a browser. This program will have to log in to your email provider's IMAP server and download all of your emails. You can use BeautifulSoup (covered in Chapter 11) to check for any instance where the word *unsubscribe* occurs within an HTML link tag.

Once you have a list of these URLs, you can use `webbrowser.open()` to automatically open all of these links in a browser.

You'll still have to manually go through and complete any additional steps to unsubscribe yourself from these lists. In most cases, this involves clicking a link to confirm.

But this script saves you from having to go through all of your emails looking for unsubscribe links. You can then pass this script along to your friends so they can run it on their email accounts. (Just make sure your email password isn't hardcoded in the source code!)

Controlling Your Computer Through Email

Write a program that checks an email account every 15 minutes for any instructions you email it and executes those instructions automatically. For example, BitTorrent is a peer-to-peer downloading system. Using free BitTorrent software such as qBittorrent, you can download large media files on your home computer. If you email the program a (completely legal, not at all piratical) BitTorrent link, the program will eventually check its email, find this message, extract the link, and then launch qBittorrent to start downloading the file. This way, you can have your home computer begin downloads while you’re away, and the (completely legal, not at all piratical) download can be finished by the time you return home.

Chapter 15 covers how to launch programs on your computer using the `subprocess.Popen()` function. For example, the following call would launch the qBittorrent program, along with a torrent file:

```
qbProcess = subprocess.Popen(['C:\\Program Files (x86)\\qBittorrent\\
qbit torrent.exe', 'shakespeare_complete_works.torrent'])
```

Of course, you’ll want the program to make sure the emails come from you. In particular, you might want to require that the emails contain a password, since it is fairly trivial for hackers to fake a “from” address in emails. The program should delete the emails it finds so that it doesn’t repeat instructions every time it checks the email account. As an extra feature, have the program email or text you a confirmation every time it executes a command. Since you won’t be sitting in front of the computer that is running the program, it’s a good idea to use the logging functions (see Chapter 10) to write a text file log that you can check if errors come up.

qBittorrent (as well as other BitTorrent applications) has a feature where it can quit automatically after the download completes. Chapter 15 explains how you can determine when a launched application has quit with the `wait()` method for `Popen` objects. The `wait()` method call will block until qBittorrent has stopped, and then your program can email or text you a notification that the download has completed.

There are a lot of possible features you could add to this project. If you get stuck, you can download an example implementation of this program from <http://nostarch.com/automatestuff/>.

17

MANIPULATING IMAGES



If you have a digital camera or even if you just upload photos from your phone to Facebook, you probably cross paths with digital image files all the time. You may know how to use basic graphics software, such as Microsoft Paint or Paintbrush, or even more advanced applications such as Adobe Photoshop. But if you need to edit a massive number of images, editing them by hand can be a lengthy, boring job.

Enter Python. Pillow is a third-party Python module for interacting with image files. The module has several functions that make it easy to crop, resize, and edit the content of an image. With the power to manipulate images the same way you would with software such as Microsoft Paint or Adobe Photoshop, Python can automatically edit hundreds or thousands of images with ease.

Computer Image Fundamentals

In order to manipulate an image, you need to understand the basics of how computers deal with colors and coordinates in images and how you can work with colors and coordinates in Pillow. But before you continue, install the `pillow` module. See Appendix A for help installing third-party modules.

Colors and *RGBA* Values

Computer programs often represent a color in an image as an *RGBA value*. An RGBA value is a group of numbers that specify the amount of red, green, blue, and *alpha* (or transparency) in a color. Each of these component values is an integer from 0 (none at all) to 255 (the maximum). These RGBA values are assigned to individual *pixels*; a pixel is the smallest dot of a single color the computer screen can show (as you can imagine, there are millions of pixels on a screen). A pixel's RGB setting tells it precisely what shade of color it should display. Images also have an alpha value to create RGBA values. If an image is displayed on the screen over a background image or desktop wallpaper, the alpha value determines how much of the background you can “see through” the image’s pixel.

In Pillow, RGBA values are represented by a tuple of four integer values. For example, the color red is represented by (255, 0, 0, 255). This color has the maximum amount of red, no green or blue, and the maximum alpha value, meaning it is fully opaque. Green is represented by (0, 255, 0, 255), and blue is (0, 0, 255, 255). White, the combination of all colors, is (255, 255, 255, 255), while black, which has no color at all, is (0, 0, 0, 255).

If a color has an alpha value of 0, it is invisible, and it doesn’t really matter what the RGB values are. After all, invisible red looks the same as invisible black.

Pillow uses the standard color names that HTML uses. Table 17-1 lists a selection of standard color names and their values.

Table 17-1: Standard Color Names and Their RGBA Values

Name	RGBA value	Name	RGBA value
White	(255, 255, 255, 255)	Red	(255, 0, 0, 255)
Green	(0, 128, 0, 255)	Blue	(0, 0, 255, 255)
Gray	(128, 128, 128, 255)	Yellow	(255, 255, 0, 255)
Black	(0, 0, 0, 255)	Purple	(128, 0, 128, 255)

Pillow offers the `ImageColor.getcolor()` function so you don’t have to memorize RGBA values for the colors you want to use. This function takes a color name string as its first argument, and the string ‘RGBA’ as its second argument, and it returns an RGBA tuple.

CMYK AND RGB COLORING

In grade school you learned that mixing red, yellow, and blue paints can form other colors; for example, you can mix blue and yellow to make green paint. This is known as the *subtractive color model*, and it applies to dyes, inks, and pigments. This is why color printers have CMYK ink cartridges: the *Cyan* (blue), *Magenta* (red), *Yellow*, and *black* ink can be mixed together to form any color.

However, the physics of light uses what's called an *additive color model*. When combining light (such as the light given off by your computer screen), red, green, and blue light can be combined to form any other color. This is why *RGB* values represent color in computer programs.

To see how this function works, enter the following into the interactive shell:

```
❶ >>> from PIL import ImageColor
❷ >>> ImageColor.getcolor('red', 'RGBA')
(255, 0, 0, 255)
❸ >>> ImageColor.getcolor('RED', 'RGBA')
(255, 0, 0, 255)
>>> ImageColor.getcolor('Black', 'RGBA')
(0, 0, 0, 255)
>>> ImageColor.getcolor('chocolate', 'RGBA')
(210, 105, 30, 255)
>>> ImageColor.getcolor('CornflowerBlue', 'RGBA')
(100, 149, 237, 255)
```

First, you need to import the `ImageColor` module from `PIL` ❶ (not from `Pillow`; you'll see why in a moment). The color name string you pass to `ImageColor.getcolor()` is case insensitive, so passing '`red`' ❷ and passing '`RED`' ❸ give you the same `RGBA` tuple. You can also pass more unusual color names, like '`chocolate`' and '`Cornflower Blue`'.

`Pillow` supports a huge number of color names, from '`aliceblue`' to '`whitesmoke`'. You can find the full list of more than 100 standard color names in the resources at <http://nostarch.com/automatestuff/>.

Coordinates and Box Tuples

Image pixels are addressed with x- and y-coordinates, which respectively specify a pixel's horizontal and vertical location in an image. The *origin* is the pixel at the top-left corner of the image and is specified with the notation `(0, 0)`. The first zero represents the x-coordinate, which starts at zero at the origin and increases going from left to right. The second zero represents the y-coordinate, which starts at zero at the origin and increases going

down the image. This bears repeating: y-coordinates increase going downward, which is the opposite of how you may remember y-coordinates being used in math class. Figure 17-1 demonstrates how this coordinate system works.

Many of Pillow’s functions and methods take a *box tuple* argument. This means Pillow is expecting a tuple of four integer coordinates that represent a rectangular region in an image. The four integers are, in order, as follows:

- *Left*: The x-coordinate of the leftmost edge of the box.
- *Top*: The y-coordinate of the top edge of the box.
- *Right*: The x-coordinate of one pixel to the right of the rightmost edge of the box. This integer must be greater than the left integer.
- *Bottom*: The y-coordinate of one pixel lower than the bottom edge of the box. This integer must be greater than the top integer.

Note that the box includes the left and top coordinates and goes up to but does not include the right and bottom coordinates. For example, the box tuple (3, 1, 9, 6) represents all the pixels in the black box in Figure 17-2.

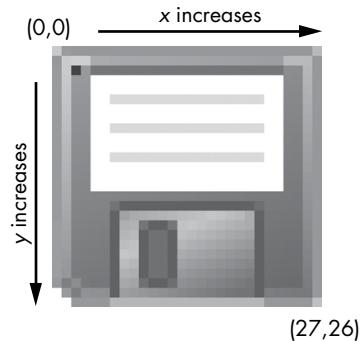


Figure 17-1: The x- and y-coordinates of a 27x26 image of some sort of ancient data storage device



Figure 17-2: The area represented by the box tuple (3, 1, 9, 6)

Manipulating Images with Pillow

Now that you know how colors and coordinates work in Pillow, let’s use Pillow to manipulate an image. Figure 17-3 is the image that will be used for all the interactive shell examples in this chapter. You can download it from <http://nostarch.com/automatestuff/>.

Once you have the image file *Zophie.png* in your current working directory, you’ll be ready to load the image of Zophie into Python, like so:

```
>>> from PIL import Image  
>>> catIm = Image.open('zophie.png')
```



Figure 17-3: My cat Zophie. The camera adds 10 pounds (which is a lot for a cat).

To load the image, you import the `Image` module from Pillow and call `Image.open()`, passing it the image's filename. You can then store the loaded image in a variable like `CatIm`. The module name of Pillow is `PIL` to make it backward compatible with an older module called Python Imaging Library, which is why you must run `from PIL import Image` instead of `from Pillow import Image`. Because of the way Pillow's creators set up the `pillow` module, you must use the `from PIL import Image` form of `import` statement, rather than simply `import PIL`.

If the image file isn't in the current working directory, change the working directory to the folder that contains the image file by calling the `os.chdir()` function.

```
>>> import os  
>>> os.chdir('C:\\\\folder_with_image_file')
```

The `Image.open()` function returns a value of the `Image` object data type, which is how Pillow represents an image as a Python value. You can load an `Image` object from an image file (of any format) by passing the `Image.open()` function a string of the filename. Any changes you make to the `Image` object can be saved to an image file (also of any format) with the `save()` method. All the rotations, resizing, cropping, drawing, and other image manipulations will be done through method calls on this `Image` object.

To shorten the examples in this chapter, I'll assume you've imported Pillow's `Image` module and that you have the `Zophie` image stored in a variable named `catIm`. Be sure that the `zophie.png` file is in the current working directory so that the `Image.open()` function can find it. Otherwise, you will also have to specify the full absolute path in the string argument to `Image.open()`.

Working with the Image Data Type

An `Image` object has several useful attributes that give you basic information about the image file it was loaded from: its width and height, the filename, and the graphics format (such as JPEG, GIF, or PNG).

For example, enter the following into the interactive shell:

```
>>> from PIL import Image
>>> catIm = Image.open('zophie.png')
>>> catIm.size
❶ (816, 1088)
❷ >>> width, height = catIm.size
❸ >>> width
816
❹ >>> height
1088
>>> catIm.filename
'zophie.png'
>>> catIm.format
'PNG'
>>> catIm.format_description
'Portable network graphics'
❺ >>> catIm.save('zophie.jpg')
```

After making an `Image` object from `Zophie.png` and storing the `Image` object in `catIm`, we can see that the object's `size` attribute contains a tuple of the image's width and height in pixels ❶. We can assign the values in the tuple to `width` and `height` variables ❷ in order to access with `width` ❸ and `height` ❹ individually. The `filename` attribute describes the original file's name. The `format` and `format_description` attributes are strings that describe the image format of the original file (with `format_description` being a bit more verbose).

Finally, calling the `save()` method and passing it '`zophie.jpg`' saves a new image with the filename `zophie.jpg` to your hard drive ❺. Pillow sees that the file extension is `.jpg` and automatically saves the image using the JPEG image format. Now you should have two images, `zophie.png` and `zophie.jpg`, on your hard drive. While these files are based on the same image, they are not identical because of their different formats.

Pillow also provides the `Image.new()` function, which returns an `Image` object—much like `Image.open()`, except the image represented by `Image.new()`'s object will be blank. The arguments to `Image.new()` are as follows:

- The string '`RGBA`', which sets the color mode to `RGBA`. (There are other modes that this book doesn't go into.)
- The size, as a two-integer tuple of the new image's width and height.

- The background color that the image should start with, as a four-integer tuple of an RGBA value. You can use the return value of the `ImageColor.getcolor()` function for this argument. Alternatively, `Image.new()` also supports just passing the string of the standard color name.

For example, enter the following into the interactive shell:

```
>>> from PIL import Image
❶ >>> im = Image.new('RGBA', (100, 200), 'purple')
>>> im.save('purpleImage.png')
❷ >>> im2 = Image.new('RGBA', (20, 20))
>>> im2.save('transparentImage.png')
```

Here we create an `Image` object for an image that's 100 pixels wide and 200 pixels tall, with a purple background ❶. This image is then saved to the file `purpleImage.png`. We call `Image.new()` again to create another `Image` object, this time passing `(20, 20)` for the dimensions and nothing for the background color ❷. Invisible black, `(0, 0, 0, 0)`, is the default color used if no color argument is specified, so the second image has a transparent background; we save this 20×20 transparent square in `transparentImage.png`.

Cropping Images

Cropping an image means selecting a rectangular region inside an image and removing everything outside the rectangle. The `crop()` method on `Image` objects takes a box tuple and returns an `Image` object representing the cropped image. The cropping does not happen in place—that is, the original `Image` object is left untouched, and the `crop()` method returns a new `Image` object. Remember that a boxed tuple—in this case, the cropped section—includes the left column and top row of pixels but only goes up to and does *not* include the right column and bottom row of pixels.

Enter the following into the interactive shell:

```
>>> croppedIm = catIm.crop((335, 345, 565, 560))
>>> croppedIm.save('cropped.png')
```

This makes a new `Image` object for the cropped image, stores the object in `croppedIm`, and then calls `save()` on `croppedIm` to save the cropped image in `cropped.png`. The new file `cropped.png` will be created from the original image, like in Figure 17-4.



Figure 17-4: The new image will be just the cropped section of the original image.

Copying and Pasting Images onto Other Images

The `copy()` method will return a new `Image` object with the same image as the `Image` object it was called on. This is useful if you need to make changes to an image but also want to keep an untouched version of the original. For example, enter the following into the interactive shell:

```
>>> catIm = Image.open('zophie.png')
>>> catCopyIm = catIm.copy()
```

The `catIm` and `catCopyIm` variables contain two separate `Image` objects, which both have the same image on them. Now that you have an `Image` object stored in `catCopyIm`, you can modify `catCopyIm` as you like and save it to a new filename, leaving `zophie.png` untouched. For example, let's try modifying `catCopyIm` with the `paste()` method.

The `paste()` method is called on an `Image` object and pastes another image on top of it. Let's continue the shell example by pasting a smaller image onto `catCopyIm`.

```
>>> faceIm = catIm.crop((335, 345, 565, 560))
>>> faceIm.size
(230, 215)
>>> catCopyIm.paste(faceIm, (0, 0))
>>> catCopyIm.paste(faceIm, (400, 500))
>>> catCopyIm.save('pasted.png')
```

First we pass `crop()` a box tuple for the rectangular area in `zophie.png` that contains Zophie’s face. This creates an `Image` object representing a 230×215 crop, which we store in `faceIm`. Now we can paste `faceIm` onto `catCopyIm`. The `paste()` method takes two arguments: a “source” `Image` object and a tuple of the x- and y-coordinates where you want to paste the top-left corner of the source `Image` object onto the main `Image` object. Here we call `paste()` twice on `catCopyIm`, passing $(0, 0)$ the first time and $(400, 500)$ the second time. This pastes `faceIm` onto `catCopyIm` twice: once with the top-left corner of `faceIm` at $(0, 0)$ on `catCopyIm`, and once with the top-left corner of `faceIm` at $(400, 500)$. Finally, we save the modified `catCopyIm` to `pasted.png`. The `pasted.png` image looks like Figure 17-5.

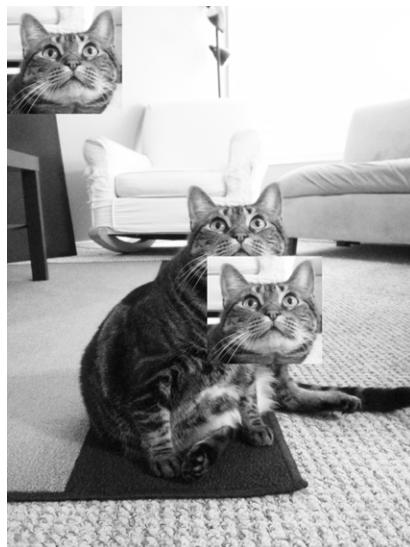


Figure 17-5: Zophie the cat, with her face pasted twice

NOTE

Despite their names, the `copy()` and `paste()` methods in Pillow do not use your computer’s clipboard.

Note that the `paste()` method modifies its `Image` object *in place*; it does not return an `Image` object with the pasted image. If you want to call `paste()` but also keep an untouched version of the original image around, you’ll need to first copy the image and then call `paste()` on that copy.

Say you want to tile Zophie’s head across the entire image, as in Figure 17-6. You can achieve this effect with just a couple `for` loops. Continue the interactive shell example by entering the following:

```
>>> catImWidth, catImHeight = catIm.size
>>> faceImWidth, faceImHeight = faceIm.size
❶ >>> catCopyTwo = catIm.copy()
```

```

❷ >>> for left in range(0, catImWidth, faceImWidth):
❸         for top in range(0, catImHeight, faceImHeight):
             print(left, top)
             catCopyTwo.paste(faceIm, (left, top))
0 0
0 215
0 430
0 645
0 860
0 1075
230 0
230 215
--snip--
690 860
690 1075
>>> catCopyTwo.save('tiled.png')

```

Here we store the width of height of `catIm` in `catImWidth` and `catImHeight`. At ❶ we make a copy of `catIm` and store it in `catCopyTwo`. Now that we have a copy that we can paste onto, we start looping to paste `faceIm` onto `catCopyTwo`. The outer for loop's `left` variable starts at 0 and increases by `faceImWidth`(230) ❷. The inner for loop's `top` variable start at 0 and increases by `faceImHeight`(215) ❸. These nested for loops produce values for `left` and `top` to paste a grid of `faceIm` images over the `catCopyTwo` `Image` object, as in Figure 17-6. To see our nested loops working, we print `left` and `top`. After the pasting is complete, we save the modified `catCopyTwo` to `tiled.png`.



Figure 17-6: Nested for loops used with `paste()` to duplicate the cat's face (a dupli-cat, if you will).

PASTING TRANSPARENT PIXELS

Normally transparent pixels are pasted as white pixels. If the image you want to paste has transparent pixels, pass the `Image` object as the third argument so that a solid rectangle isn't pasted. This third argument is the "mask" `Image` object. A mask is an `Image` object where the alpha value is significant, but the red, green, and blue values are ignored. The mask tells the `paste()` function which pixels it should copy and which it should leave transparent. Advanced usage of masks is beyond this book, but if you want to paste an image that has transparent pixels, pass the `Image` object again as the third argument.

Resizing an Image

The `resize()` method is called on an `Image` object and returns a new `Image` object of the specified width and height. It accepts a two-integer tuple argument, representing the new width and height of the returned image. Enter the following into the interactive shell:

```
❶ >>> width, height = catIm.size
❷ >>> quartersizedIm = catIm.resize((int(width / 2), int(height / 2)))
      >>> quartersizedIm.save('quartersized.png')
❸ >>> svelteIm = catIm.resize((width, height + 300))
      >>> svelteIm.save('svelte.png')
```

Here we assign the two values in the `catIm.size` tuple to the variables `width` and `height` ❶. Using `width` and `height` instead of `catIm.size[0]` and `catIm.size[1]` makes the rest of the code more readable.

The first `resize()` call passes `int(width / 2)` for the new width and `int(height / 2)` for the new height ❷, so the `Image` object returned from `resize()` will be half the length and width of the original image, or one-quarter of the original image size overall. The `resize()` method accepts only integers in its tuple argument, which is why you needed to wrap both divisions by 2 in an `int()` call.

This resizing keeps the same proportions for the width and height. But the new width and height passed to `resize()` do not have to be proportional to the original image. The `svelteIm` variable contains an `Image` object that has the original width but a height that is 300 pixels taller ❸, giving Zophie a more slender look.

Note that the `resize()` method does not edit the `Image` object in place but instead returns a new `Image` object.

Rotating and Flipping Images

Images can be rotated with the `rotate()` method, which returns a new `Image` object of the rotated image and leaves the original `Image` object unchanged. The argument to `rotate()` is a single integer or float representing the number of degrees to rotate the image counterclockwise. Enter the following into the interactive shell:

```
>>> catIm.rotate(90).save('rotated90.png')
>>> catIm.rotate(180).save('rotated180.png')
>>> catIm.rotate(270).save('rotated270.png')
```

Note how you can *chain* method calls by calling `save()` directly on the `Image` object returned from `rotate()`. The first `rotate()` and `save()` call makes a new `Image` object representing the image rotated counterclockwise by 90 degrees and saves the rotated image to `rotated90.png`. The second and third calls do the same, but with 180 degrees and 270 degrees. The results look like Figure 17-7.



Figure 17-7: The original image (left) and the image rotated counterclockwise by 90, 180, and 270 degrees

Notice that the width and height of the image change when the image is rotated 90 or 270 degrees. If you rotate an image by some other amount, the original dimensions of the image are maintained. On Windows, a black background is used to fill in any gaps made by the rotation, like in Figure 17-8. On OS X, transparent pixels are used for the gaps instead.

The `rotate()` method has an optional `expand` keyword argument that can be set to `True` to enlarge the dimensions of the image to fit the entire rotated new image. For example, enter the following into the interactive shell:

```
>>> catIm.rotate(6).save('rotated6.png')
>>> catIm.rotate(6, expand=True).save('rotated6_expanded.png')
```

The first call rotates the image 6 degrees and saves it to `rotate6.png` (see the image on the left of Figure 17-8). The second call rotates the image 6 degrees with `expand` set to `True` and saves it to `rotate6_expanded.png` (see the image on the right of Figure 17-8).



Figure 17-8: The image rotated 6 degrees normally (left) and with `expand=True` (right)

You can also get a “mirror flip” of an image with the `transpose()` method. You must pass either `Image.FLIP_LEFT_RIGHT` or `Image.FLIP_TOP_BOTTOM` to the `transpose()` method. Enter the following into the interactive shell:

```
>>> catIm.transpose(Image.FLIP_LEFT_RIGHT).save('horizontal_flip.png')
>>> catIm.transpose(Image.FLIP_TOP_BOTTOM).save('vertical_flip.png')
```

Like `rotate()`, `transpose()` creates a new `Image` object. Here we pass `Image.FLIP_LEFT_RIGHT` to flip the image horizontally and then save the result to `horizontal_flip.png`. To flip the image vertically, we pass `Image.FLIP_TOP_BOTTOM` and save to `vertical_flip.png`. The results look like Figure 17-9.

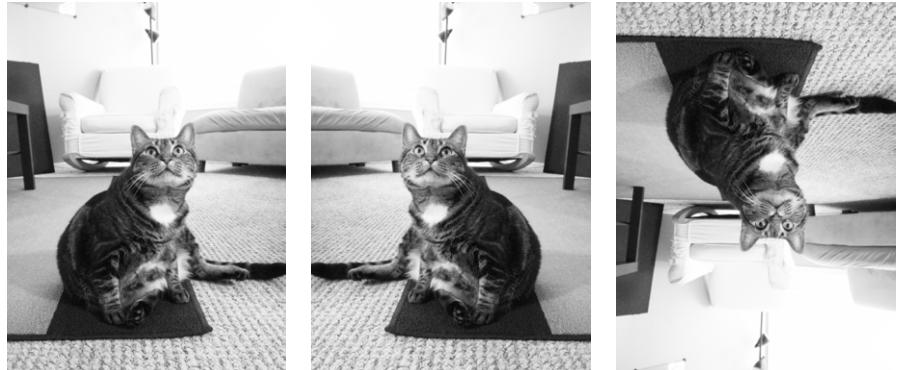


Figure 17-9: The original image (left), horizontal flip (center), and vertical flip (right)

Changing Individual Pixels

The color of an individual pixel can be retrieved or set with the `getpixel()` and `putpixel()` methods. These methods both take a tuple representing the x- and y-coordinates of the pixel. The `putpixel()` method also takes an additional tuple argument for the color of the pixel. This color argument is a four-integer RGBA tuple or a three-integer RGB tuple. Enter the following into the interactive shell:

```
❶ >>> im = Image.new('RGBA', (100, 100))
❷ >>> im.getpixel((0, 0))
(0, 0, 0, 0)
❸ >>> for x in range(100):
    for y in range(50):
        im.putpixel((x, y), (210, 210, 210))

    >>> from PIL import ImageColor
❹ >>> for x in range(100):
    for y in range(50, 100):
        im.putpixel((x, y), ImageColor.getcolor('darkgray', 'RGBA'))
    >>> im.getpixel((0, 0))
(210, 210, 210, 255)
    >>> im.getpixel((0, 50))
(169, 169, 169, 255)
    >>> im.save('putPixel.png')
```

At ❶ we make a new image that is a 100×100 transparent square. Calling `getpixel()` on some coordinates in this image returns `(0, 0, 0, 0)` because the image is transparent ❷. To color pixels in this image, we can use nested `for` loops to go through all the pixels in the top half of the image ❸ and color each pixel using `putpixel()` ❹. Here we pass `putpixel()` the RGB tuple `(210, 210, 210)`, a light gray.

Say we want to color the bottom half of the image dark gray but don't know the RGB tuple for dark gray. The `putpixel()` method doesn't accept a standard color name like `'darkgray'`, so you have to use `ImageColor.getcolor()` to get a color tuple from `'darkgray'`. Loop through the pixels in the bottom half of the image ❺ and pass `putpixel()` the return value of `ImageColor.getcolor()` ❻, and you should now have an image that is light gray in its top half and dark gray in the bottom half, as shown in Figure 17-10. You can call `getpixel()` on some coordinates to confirm that the color at any given pixel is what you expect. Finally, save the image to `putPixel.png`.

Of course, drawing one pixel at a time onto an image isn't very convenient. If you need to draw shapes, use the `ImageDraw` functions explained later in this chapter.



Figure 17-10: The `putPixel.png` image

Project: Adding a Logo

Say you have the boring job of resizing thousands of images and adding a small logo watermark to the corner of each. Doing this with a basic graphics program such as Paintbrush or Paint would take forever. A fancier graphics application such as Photoshop can do batch processing, but that software costs hundreds of dollars. Let's write a script to do it instead.

Say that Figure 17-11 is the logo you want to add to the bottom-right corner of each image: a black cat icon with a white border, with the rest of the image transparent.

At a high level, here's what the program should do:

- Load the logo image.
- Loop over all `.png` and `.jpg` files in the working directory.
- Check whether the image is wider or taller than 300 pixels.
- If so, reduce the width or height (whichever is larger) to 300 pixels and scale down the other dimension proportionally.
- Paste the logo image into the corner.
- Save the altered images to another folder.

This means the code will need to do the following:

- Open the `catlogo.png` file as an `Image` object.
- Loop over the strings returned from `os.listdir('.')`.
- Get the width and height of the image from the `size` attribute.
- Calculate the new width and height of the resized image.
- Call the `resize()` method to resize the image.
- Call the `paste()` method to paste the logo.
- Call the `save()` method to save the changes, using the original filename.



Figure 17-11: The logo to be added to the image.

Step 1: Open the Logo Image

For this project, open a new file editor window, enter the following code, and save it as `resizeAndAddLogo.py`:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.
```

```
import os
from PIL import Image

❶ SQUARE_FIT_SIZE = 300
❷ LOGO_FILENAME = 'catlogo.png'

❸ logoIm = Image.open(LOGO_FILENAME)
❹ logoWidth, logoHeight = logoIm.size

# TODO: Loop over all files in the working directory.

# TODO: Check if image needs to be resized.

# TODO: Calculate the new width and height to resize to.

# TODO: Resize the image.

# TODO: Add the logo.

# TODO: Save changes.
```

By setting up the `SQUARE_FIT_SIZE` ❶ and `LOGO_FILENAME` ❷ constants at the start of the program, we've made it easy to change the program later. Say the logo that you're adding isn't the cat icon, or say you're reducing the output images' largest dimension to something other than 300 pixels. With these constants at the start of the program, you can just open the code, change those values once, and you're done. (Or you can make it so that the values for these constants are taken from the command line arguments.) Without these constants, you'd instead have to search the code for all instances of `300` and `'catlogo.png'` and replace them with the values for your new project. In short, using constants makes your program more generalized.

The logo `Image` object is returned from `Image.open()` ❸. For readability, `logoWidth` and `logoHeight` are assigned to the values from `logoIm.size` ❹.

The rest of the program is a skeleton of TODO comments for now.

Step 2: Loop Over All Files and Open Images

Now you need to find every `.png` file and `.jpg` file in the current working directory. Note that you don't want to add the logo image to the logo image itself, so the program should skip any image with a filename that's the same as `LOGO_FILENAME`. Add the following to your code:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image

--snip--
```

```
os.makedirs('withLogo', exist_ok=True)
# Loop over all files in the working directory.
❶ for filename in os.listdir('.'):
❷     if not (filename.endswith('.png') or filename.endswith('.jpg')) \
        or filename == LOGO_FILENAME:
❸     continue    # skip non-image files and the logo file itself

❹     im = Image.open(filename)
     width, height = im.size
```

--snip--

First, the `os.makedirs()` call creates a `withLogo` folder to store the finished images with logos, instead of overwriting the original image files. The `exist_ok=True` keyword argument will keep `os.makedirs()` from raising an exception if `withLogo` already exists. While looping through all the files in the working directory with `os.listdir('.')` ❶, the long `if` statement ❷ checks whether each filename doesn't end with `.png` or `.jpg`. If so—or if the file is the logo image itself—then the loop should skip it and use `continue` ❸ to go to the next file. If `filename` *does* end with `'.png'` or `'.jpg'` (and isn't the logo file), you can open it as an `Image` object ❹ and set `width` and `height`.

Step 3: Resize the Images

The program should resize the image only if the width or height is larger than `SQUARE_FIT_SIZE` (300 pixels, in this case), so put all of the resizing code inside an `if` statement that checks the `width` and `height` variables. Add the following code to your program:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image

--snip--

# Check if image needs to be resized.
if width > SQUARE_FIT_SIZE and height > SQUARE_FIT_SIZE:
    # Calculate the new width and height to resize to.
    if width > height:
❶        height = int((SQUARE_FIT_SIZE / width) * height)
        width = SQUARE_FIT_SIZE
    else:
❷        width = int((SQUARE_FIT_SIZE / height) * width)
        height = SQUARE_FIT_SIZE

    # Resize the image.
    print('Resizing %s...' % (filename))
❸    im = im.resize((width, height))

--snip--
```

If the image does need to be resized, you need to find out whether it is a wide or tall image. If width is greater than height, then the height should be reduced by the same proportion that the width would be reduced ❶. This proportion is the `SQUARE_FIT_SIZE` value divided by the current width. The new height value is this proportion multiplied by the current height value. Since the division operator returns a float value and `resize()` requires the dimensions to be integers, remember to convert the result to an integer with the `int()` function. Finally, the new width value will simply be set to `SQUARE_FIT_SIZE`.

If the height is greater than or equal to the width (both cases are handled in the else clause), then the same calculation is done, except with the height and width variables swapped ❷.

Once width and height contain the new image dimensions, pass them to the `resize()` method and store the returned `Image` object in `im` ❸.

Step 4: Add the Logo and Save the Changes

Whether or not the image was resized, the logo should still be pasted to the bottom-right corner. Where exactly the logo should be pasted depends on both the size of the image and the size of the logo. Figure 17-12 shows how to calculate the pasting position. The left coordinate for where to paste the logo will be the image width minus the logo width; the top coordinate for where to paste the logo will be the image height minus the logo height.



Figure 17-12: The left and top coordinates for placing the logo in the bottom-right corner should be the image width/height minus the logo width/height.

After your code pastes the logo into the image, it should save the modified `Image` object. Add the following to your program:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image
```

```
--snip--
```

```
# Check if image needs to be resized.  
--snip--  
  
❶ # Add the logo.  
❷ print('Adding logo to %s...' % (filename))  
❸ im.paste(logoIm, (width - logoWidth, height - logoHeight), logoIm)  
  
❹ # Save changes.  
❺ im.save(os.path.join('withLogo', filename))
```

The new code prints a message telling the user that the logo is being added ❶, pastes `logoIm` onto `im` at the calculated coordinates ❷, and saves the changes to a filename in the `withLogo` directory ❸. When you run this program with the `zophie.png` file as the only image in the working directory, the output will look like this:

```
Resizing zophie.png...  
Adding logo to zophie.png...
```

The image `zophie.png` will be changed to a 225×300 -pixel image that looks like Figure 17-13. Remember that the `paste()` method will not paste the transparency pixels if you do not pass the `logoIm` for the third argument as well. This program can automatically resize and “logo-ify” hundreds of images in just a couple minutes.



Figure 17-13: The image `zophie.png` resized and the logo added (left). If you forget the third argument, the transparent pixels in the logo will be copied as solid white pixels (right).

Ideas for Similar Programs

Being able to composite images or modify image sizes in a batch can be useful in many applications. You could write similar programs to do the following:

- Add text or a website URL to images.
- Add timestamps to images.
- Copy or move images into different folders based on their sizes.
- Add a mostly transparent watermark to an image to prevent others from copying it.

Drawing on Images

If you need to draw lines, rectangles, circles, or other simple shapes on an image, use Pillow's `ImageDraw` module. Enter the following into the interactive shell:

```
>>> from PIL import Image, ImageDraw
>>> im = Image.new('RGBA', (200, 200), 'white')
>>> draw = ImageDraw.Draw(im)
```

First, we import `Image` and `ImageDraw`. Then we create a new image, in this case, a 200×200 white image, and store the `Image` object in `im`. We pass the `Image` object to the `ImageDraw.Draw()` function to receive an `ImageDraw` object. This object has several methods for drawing shapes and text onto an `Image` object. Store the `ImageDraw` object in a variable like `draw` so you can use it easily in the following example.

Drawing Shapes

The following `ImageDraw` methods draw various kinds of shapes on the image. The `fill` and `outline` parameters for these methods are optional and will default to white if left unspecified.

Points

The `point(xy, fill)` method draws individual pixels. The `xy` argument represents a list of the points you want to draw. The list can be a list of x- and y-coordinate tuples, such as `[(x, y), (x, y), ...]`, or a list of x- and y-coordinates without tuples, such as `[x1, y1, x2, y2, ...]`. The `fill` argument is the color of the points and is either an RGBA tuple or a string of a color name, such as `'red'`. The `fill` argument is optional.

Lines

The `line(xy, fill, width)` method draws a line or series of lines. `xy` is either a list of tuples, such as `[(x, y), (x, y), ...]`, or a list of integers, such as `[x1, y1, x2, y2, ...]`. Each point is one of the connecting points on the

lines you’re drawing. The optional *fill* argument is the color of the lines, as an RGBA tuple or color name. The optional *width* argument is the width of the lines and defaults to 1 if left unspecified.

Rectangles

The `rectangle(xy, fill, outline)` method draws a rectangle. The *xy* argument is a box tuple of the form `(left, top, right, bottom)`. The *left* and *top* values specify the x- and y-coordinates of the upper-left corner of the rectangle, while *right* and *bottom* specify the lower-right corner. The optional *fill* argument is the color that will fill the inside of the rectangle. The optional *outline* argument is the color of the rectangle’s outline.

Ellipses

The `ellipse(xy, fill, outline)` method draws an ellipse. If the width and height of the ellipse are identical, this method will draw a circle. The *xy* argument is a box tuple `(left, top, right, bottom)` that represents a box that precisely contains the ellipse. The optional *fill* argument is the color of the inside of the ellipse, and the optional *outline* argument is the color of the ellipse’s outline.

Polygons

The `polygon(xy, fill, outline)` method draws an arbitrary polygon. The *xy* argument is a list of tuples, such as `[(x, y), (x, y), ...]`, or integers, such as `[x1, y1, x2, y2, ...]`, representing the connecting points of the polygon’s sides. The last pair of coordinates will be automatically connected to the first pair. The optional *fill* argument is the color of the inside of the polygon, and the optional *outline* argument is the color of the polygon’s outline.

Drawing Example

Enter the following into the interactive shell:

```
>>> from PIL import Image, ImageDraw
>>> im = Image.new('RGBA', (200, 200), 'white')
>>> draw = ImageDraw.Draw(im)
❶ >>> draw.line([(0, 0), (199, 0), (199, 199), (0, 199), (0, 0)], fill='black')
❷ >>> draw.rectangle((20, 30, 60, 60), fill='blue')
❸ >>> draw.ellipse((120, 30, 160, 60), fill='red')
❹ >>> draw.polygon(((57, 87), (79, 62), (94, 85), (120, 90), (103, 113)),
    fill='brown')
❺ >>> for i in range(100, 200, 10):
    draw.line([(i, 0), (200, i - 100)], fill='green')

>>> im.save('drawing.png')
```

After making an `Image` object for a 200×200 white image, passing it to `ImageDraw.Draw()` to get an `ImageDraw` object, and storing the `ImageDraw` object in `draw`, you can call drawing methods on `draw`. Here we make a thin, black

outline at the edges of the image ❶, a blue rectangle with its top-left corner at (20, 30) and bottom-right corner at (60, 60) ❷, a red ellipse defined by a box from (120, 30) to (160, 60) ❸, a brown polygon with five points ❹, and a pattern of green lines drawn with a for loop ❺. The resulting *drawing.png* file will look like Figure 17-14.



Figure 17-14: The resulting *drawing.png* image

There are several other shape-drawing methods for `ImageDraw` objects. The full documentation is available at <http://pillow.readthedocs.org/en/latest/reference/ImageDraw.html>.

Drawing Text

The `ImageDraw` object also has a `text()` method for drawing text onto an image. The `text()` method takes four arguments: `xy`, `text`, `fill`, and `font`.

- The `xy` argument is a two-integer tuple specifying the upper-left corner of the text box.
- The `text` argument is the string of text you want to write.
- The optional `fill` argument is the color of the text.
- The optional `font` argument is an `ImageFont` object, used to set the typeface and size of the text. This is described in more detail in the next section.

Since it's often hard to know in advance what size a block of text will be in a given font, the `ImageDraw` module also offers a `textsize()` method. Its first argument is the string of text you want to measure, and its second argument is an optional `ImageFont` object. The `textsize()` method will then return a two-integer tuple of the width and height that the text in the given

font would be if it were written onto the image. You can use this width and height to help you calculate exactly where you want to put the text on your image.

The first three arguments for `text()` are straightforward. Before we use `text()` to draw text onto an image, let's look at the optional fourth argument, the `ImageFont` object.

Both `text()` and `textsize()` take an optional `ImageFont` object as their final arguments. To create one of these objects, first run the following:

```
>>> from PIL import ImageFont
```

Now that you've imported Pillow's `ImageFont` module, you can call the `ImageFont.truetype()` function, which takes two arguments. The first argument is a string for the font's *TrueType file*—this is the actual font file that lives on your hard drive. A TrueType file has the *.ttf* file extension and can usually be found in the following folders:

- On Windows: *C:\Windows\Fonts*
- On OS X: */Library/Fonts* and */System/Library/Fonts*
- On Linux: */usr/share/fonts/truetype*

You don't actually need to enter these paths as part of the TrueType file string because Python knows to automatically search for fonts in these directories. But Python will display an error if it is unable to find the font you specified.

The second argument to `ImageFont.truetype()` is an integer for the font size in *points* (rather than, say, pixels). Keep in mind that Pillow creates PNG images that are 72 pixels per inch by default, and a point is 1/72 of an inch.

Enter the following into the interactive shell, replacing `FONT_FOLDER` with the actual folder name your operating system uses:

```
>>> from PIL import Image, ImageDraw, ImageFont
>>> import os
❶ >>> im = Image.new('RGBA', (200, 200), 'white')
❷ >>> draw = ImageDraw.Draw(im)
❸ >>> draw.text((20, 150), 'Hello', fill='purple')
>>> fontsFolder = 'FONT_FOLDER' # e.g. '/Library/Fonts'
❹ >>> arialFont = ImageFont.truetype(os.path.join(fontsFolder, 'arial.ttf'), 32)
❺ >>> draw.text((100, 150), 'Howdy', fill='gray', font=arialFont)
>>> im.save('text.png')
```

After importing `Image`, `ImageDraw`, `ImageFont`, and `os`, we make an `Image` object for a new 200×200 white image ❶ and make an `ImageDraw` object from the `Image` object ❷. We use `text()` to draw *Hello* at (20, 150) in purple ❸. We didn't pass the optional fourth argument in this `text()` call, so the typeface and size of this text aren't customized.

To set a typeface and size, we first store the folder name (like `/Library/Fonts`) in `fontsFolder`. Then we call `ImageFont.truetype()`, passing it the `.ttf` file for the font we want, followed by an integer font size ❸. Store the `Font` object you get from `ImageFont.truetype()` in a variable like `arialFont`, and then pass the variable to `text()` in the final keyword argument. The `text()` call at ❹ draws *Howdy* at (100, 150) in gray in 32-point Arial.

The resulting `text.png` file will look like Figure 17-15.



Figure 17-15: The resulting `text.png` image

Summary

Images consist of a collection of pixels, and each pixel has an RGBA value for its color and its addressable by x- and y-coordinates. Two common image formats are JPEG and PNG. The `pillow` module can handle both of these image formats and others.

When an image is loaded into an `Image` object, its width and height dimensions are stored as a two-integer tuple in the `size` attribute. Objects of the `Image` data type also have methods for common image manipulations: `crop()`, `copy()`, `paste()`, `resize()`, `rotate()`, and `transpose()`. To save the `Image` object to an image file, call the `save()` method.

If you want your program to draw shapes onto an image, use `ImageDraw` methods to draw points, lines, rectangles, ellipses, and polygons. The module also provides methods for drawing text in a typeface and font size of your choosing.

Although advanced (and expensive) applications such as Photoshop provide automatic batch processing features, you can use Python scripts to do many of the same modifications for free. In the previous chapters, you wrote Python programs to deal with plaintext files, spreadsheets, PDFs, and other formats. With the `pillow` module, you've extended your programming powers to processing images as well!

Practice Questions

1. What is an RGBA value?
2. How can you get the RGBA value of 'CornflowerBlue' from the `Pillow` module?
3. What is a box tuple?
4. What function returns an `Image` object for, say, an image file named `zophie.png`?

5. How can you find out the width and height of an `Image` object's image?
6. What method would you call to get `Image` object for a 100×100 image, excluding the lower left quarter of it?
7. After making changes to an `Image` object, how could you save it as an image file?
8. What module contains Pillow's shape-drawing code?
9. `Image` objects do not have drawing methods. What kind of object does? How do you get this kind of object?

Practice Projects

For practice, write programs that do the following.

Extending and Fixing the Chapter Project Programs

The `resizeAndAddLogo.py` program in this chapter works with PNG and JPEG files, but Pillow supports many more formats than just these two. Extend `resizeAndAddLogo.py` to process GIF and BMP images as well.

Another small issue is that the program modifies PNG and JPEG files only if their file extensions are set in lowercase. For example, it will process `zophie.png` but not `zophie.PNG`. Change the code so that the file extension check is case insensitive.

Finally, the logo added to the bottom-right corner is meant to be just a small mark, but if the image is about the same size as the logo itself, the result will look like Figure 17-16. Modify `resizeAndAddLogo.py` so that the image must be at least twice the width and height of the logo image before the logo is pasted. Otherwise, it should skip adding the logo.



Figure 17-16: When the image isn't much larger than the logo, the results look ugly.

Identifying Photo Folders on the Hard Drive

I have a bad habit of transferring files from my digital camera to temporary folders somewhere on the hard drive and then forgetting about these folders. It would be nice to write a program that could scan the entire hard drive and find these leftover “photo folders.”

Write a program that goes through every folder on your hard drive and finds potential photo folders. Of course, first you'll have to define what you consider a “photo folder” to be; let's say that it's any folder where more than half of the files are photos. And how do you define what files are photos?

First, a photo file must have the file extension `.png` or `.jpg`. Also, photos are large images; a photo file's width and height must both be larger than 500 pixels. This is a safe bet, since most digital camera photos are several thousand pixels in width and height.

As a hint, here's a rough skeleton of what this program might look like:

```
#! python3
# Import modules and write comments to describe this program.

for foldername, subfolders, filenames in os.walk('C:\\\\'):
    numPhotoFiles = 0
    numNonPhotoFiles = 0
    for filename in filenames:
        # Check if file extension isn't .png or .jpg.
        if TODO:
            numNonPhotoFiles += 1
            continue    # skip to next filename

        # Open image file using Pillow.

        # Check if width & height are larger than 500.
        if TODO:
            # Image is large enough to be considered a photo.
            numPhotoFiles += 1
        else:
            # Image is too small to be a photo.
            numNonPhotoFiles += 1

    # If more than half of files were photos,
    # print the absolute path of the folder.
    if TODO:
        print(TODO)
```

When the program runs, it should print the absolute path of any photo folders to the screen.

Custom Seating Cards

Chapter 13 included a practice project to create custom invitations from a list of guests in a plaintext file. As an additional project, use the `Pillow` module to create images for custom seating cards for your guests. For each of the guests listed in the `guests.txt` file from the resources at <http://nostarch.com/automatestuff/>, generate an image file with the guest name and some flowery decoration. A public domain flower image is available in the resources at <http://nostarch.com/automatestuff/>.

To ensure that each seating card is the same size, add a black rectangle on the edges of the invitation image so that when the image is printed out, there will be a guideline for cutting. The PNG files that Pillow produces are set to 72 pixels per inch, so a 4×5-inch card would require a 288×360-pixel image.

18

CONTROLLING THE KEYBOARD AND MOUSE WITH GUI AUTOMATION



Knowing various Python modules for editing spreadsheets, downloading files, and launching programs is useful, but sometimes there just aren't any modules for the applications you need to work with. The ultimate tools for automating tasks on your computer are programs you write that directly control the keyboard and mouse. These programs can control other applications by sending them virtual keystrokes and mouse clicks, just as if you were sitting at your computer and interacting with the applications yourself. This technique is known as *graphical user interface automation*, or *GUI automation* for short. With GUI automation, your programs can do anything that a human user sitting at the computer can do, except spill coffee on the keyboard.

Think of GUI automation as programming a robotic arm. You can program the robotic arm to type at your keyboard and move your mouse for you. This technique is particularly useful for tasks that involve a lot of mindless clicking or filling out of forms.

The `pyautogui` module has functions for simulating mouse movements, button clicks, and scrolling the mouse wheel. This chapter covers only a subset of PyAutoGUI's features; you can find the full documentation at <http://pyautogui.readthedocs.org/>.

Installing the `pyautogui` Module

The `pyautogui` module can send virtual keypresses and mouse clicks to Windows, OS X, and Linux. Depending on which operating system you're using, you may have to install some other modules (called *dependencies*) before you can install PyAutoGUI.

- On Windows, there are no other modules to install.
- On OS X, run `sudo pip3 install pyobjc-framework-Quartz`, `sudo pip3 install pyobjc-core`, and then `sudo pip3 install pyobjc`.
- On Linux, run `sudo pip3 install python3-xlib`, `sudo apt-get install scrot`, `sudo apt-get install python3-tk`, and `sudo apt-get install python3-dev`. (Scrot is a screenshot program that PyAutoGUI uses.)

After these dependencies are installed, run `pip install pyautogui` (or `pip3` on OS X and Linux) to install PyAutoGUI.

Appendix A has complete information on installing third-party modules. To test whether PyAutoGUI has been installed correctly, run `import pyautogui` from the interactive shell and check for any error messages.

Staying on Track

Before you jump in to a GUI automation, you should know how to escape problems that may arise. Python can move your mouse and type keystrokes at an incredible speed. In fact, it might be too fast for other programs to keep up with. Also, if something goes wrong but your program keeps moving the mouse around, it will be hard to tell what exactly the program is doing or how to recover from the problem. Like the enchanted brooms from Disney's *The Sorcerer's Apprentice*, which kept filling—and then overfilling—Mickey's tub with water, your program could get out of control even though it's following your instructions perfectly. Stopping the program can be difficult if the mouse is moving around on its own, preventing you from clicking the IDLE window to close it. Fortunately, there are several ways to prevent or recover from GUI automation problems.

Shutting Down Everything by Logging Out

Perhaps the simplest way to stop an out-of-control GUI automation program is to log out, which will shut down all running programs. On Windows and Linux, the logout hotkey is CTRL-ALT-DEL. On OS X, it is ⌘-SHIFT-OPTION-Q. By logging out, you'll lose any unsaved work, but at least you won't have to wait for a full reboot of the computer.

Pauses and Fail-Safes

You can tell your script to wait after every function call, giving you a short window to take control of the mouse and keyboard if something goes wrong. To do this, set the `pyautogui.PAUSE` variable to the number of seconds you want it to pause. For example, after setting `pyautogui.PAUSE = 1.5`, every PyAutoGUI function call will wait one and a half seconds after performing its action. Non-PyAutoGUI instructions will not have this pause.

PyAutoGUI also has a fail-safe feature. Moving the mouse cursor to the upper-left corner of the screen will cause PyAutoGUI to raise the `pyautogui.FailSafeException` exception. Your program can either handle this exception with `try` and `except` statements or let the exception crash your program. Either way, the fail-safe feature will stop the program if you quickly move the mouse as far up and left as you can. You can disable this feature by setting `pyautogui.FAILSAFE = False`. Enter the following into the interactive shell:

```
>>> import pyautogui  
>>> pyautogui.PAUSE = 1  
>>> pyautogui.FAILSAFE = True
```

Here we import `pyautogui` and set `pyautogui.PAUSE` to 1 for a one-second pause after each function call. We set `pyautogui.FAILSAFE` to `True` to enable the fail-safe feature.

Controlling Mouse Movement

In this section, you'll learn how to move the mouse and track its position on the screen using PyAutoGUI, but first you need to understand how PyAutoGUI works with coordinates.

The mouse functions of PyAutoGUI use x- and y-coordinates. Figure 18-1 shows the coordinate system for the computer screen; it's similar to the coordinate system used for images, discussed in Chapter 17. The *origin*, where x and y are both zero, is at the upper-left corner of the screen. The x-coordinates increase going to the right, and the y-coordinates increase going down. All coordinates are positive integers; there are no negative coordinates.

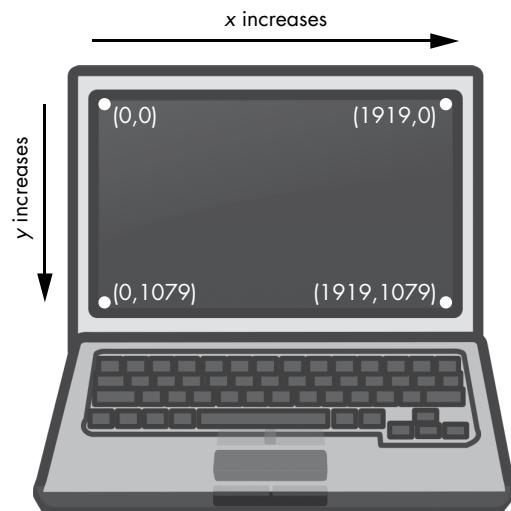


Figure 18-1: The coordinates of a computer screen with 1920x1080 resolution

Your *resolution* is how many pixels wide and tall your screen is. If your screen's resolution is set to 1920×1080, then the coordinate for the upper-left corner will be (0, 0), and the coordinate for the bottom-right corner will be (1919, 1079).

The `pyautogui.size()` function returns a two-integer tuple of the screen's width and height in pixels. Enter the following into the interactive shell:

```
>>> import pyautogui  
>>> pyautogui.size()  
(1920, 1080)  
>>> width, height = pyautogui.size()
```

`pyautogui.size()` returns (1920, 1080) on a computer with a 1920×1080 resolution; depending on your screen's resolution, your return value may be different. You can store the width and height from `pyautogui.size()` in variables like `width` and `height` for better readability in your programs.

Moving the Mouse

Now that you understand screen coordinates, let's move the mouse. The `pyautogui.moveTo()` function will instantly move the mouse cursor to a specified position on the screen. Integer values for the x- and y-coordinates make up the function's first and second arguments, respectively. An optional duration integer or float keyword argument specifies the number of seconds it should take to move the mouse to the destination. If you leave it out, the default is 0 for instantaneous movement. (All of the duration keyword arguments in PyAutoGUI functions are optional.) Enter the following into the interactive shell:

```
>>> import pyautogui  
>>> for i in range(10):  
    pyautogui.moveTo(100, 100, duration=0.25)  
    pyautogui.moveTo(200, 100, duration=0.25)  
    pyautogui.moveTo(200, 200, duration=0.25)  
    pyautogui.moveTo(100, 200, duration=0.25)
```

This example moves the mouse cursor clockwise in a square pattern among the four coordinates provided a total of ten times. Each movement takes a quarter of a second, as specified by the `duration=0.25` keyword argument. If you hadn't passed a third argument to any of the `pyautogui.moveTo()` calls, the mouse cursor would have instantly teleported from point to point.

The `pyautogui.moveRel()` function moves the mouse cursor *relative* to its current position. The following example moves the mouse in the same square pattern, except it begins the square from wherever the mouse happens to be on the screen when the code starts running:

```
>>> import pyautogui  
>>> for i in range(10):  
    pyautogui.moveRel(100, 0, duration=0.25)  
    pyautogui.moveRel(0, 100, duration=0.25)
```

```
pyautogui.moveRel(-100, 0, duration=0.25)
pyautogui.moveRel(0, -100, duration=0.25)
```

`pyautogui.moveRel()` also takes three arguments: how many pixels to move horizontally to the right, how many pixels to move vertically downward, and (optionally) how long it should take to complete the movement. A negative integer for the first or second argument will cause the mouse to move left or upward, respectively.

Getting the Mouse Position

You can determine the mouse's current position by calling the `pyautogui.position()` function, which will return a tuple of the mouse cursor's *x* and *y* positions at the time of the function call. Enter the following into the interactive shell, moving the mouse around after each call:

```
>>> pyautogui.position()
(311, 622)
>>> pyautogui.position()
(377, 481)
>>> pyautogui.position()
(1536, 637)
```

Of course, your return values will vary depending on where your mouse cursor is.

Project: “Where Is the Mouse Right Now?”

Being able to determine the mouse position is an important part of setting up your GUI automation scripts. But it's almost impossible to figure out the exact coordinates of a pixel just by looking at the screen. It would be handy to have a program that constantly displays the *x*- and *y*-coordinates of the mouse cursor as you move it around.

At a high level, here's what your program should do:

- Display the current *x*- and *y*-coordinates of the mouse cursor.
- Update these coordinates as the mouse moves around the screen.

This means your code will need to do the following:

- Call the `position()` function to fetch the current coordinates.
- Erase the previously printed coordinates by printing \b backspace characters to the screen.
- Handle the `KeyboardInterrupt` exception so the user can press CTRL-C to quit.

Open a new file editor window and save it as `mouseNow.py`.

Step 1: Import the Module

Start your program with the following:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
import pyautogui
print('Press Ctrl-C to quit.')
#TODO: Get and print the mouse coordinates.
```

The beginning of the program imports the `pyautogui` module and prints a reminder to the user that they have to press CTRL-C to quit.

Step 2: Set Up the Quit Code and Infinite Loop

You can use an infinite `while` loop to constantly print the current mouse coordinates from `mouse.position()`. As for the code that quits the program, you'll need to catch the `KeyboardInterrupt` exception, which is raised whenever the user presses CTRL-C. If you don't handle this exception, it will display an ugly traceback and error message to the user. Add the following to your program:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
import pyautogui
print('Press Ctrl-C to quit.')
try:
    while True:
        # TODO: Get and print the mouse coordinates.
❶ except KeyboardInterrupt:
❷     print('\nDone.'
```

To handle the exception, enclose the infinite `while` loop in a `try` statement. When the user presses CTRL-C, the program execution will move to the `except` clause ❶ and `Done.` will be printed in a new line ❷.

Step 3: Get and Print the Mouse Coordinates

The code inside the `while` loop should get the current mouse coordinates, format them to look nice, and print them. Add the following code to the inside of the `while` loop:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
import pyautogui
print('Press Ctrl-C to quit.')
--snip--
    # Get and print the mouse coordinates.
    x, y = pyautogui.position()
    positionStr = 'X: ' + str(x).rjust(4) + ' Y: ' + str(y).rjust(4)
--snip--
```

Using the multiple assignment trick, the `x` and `y` variables are given the values of the two integers returned in the tuple from `pyautogui.position()`. By passing `x` and `y` to the `str()` function, you can get string forms of the integer coordinates. The `rjust()` string method will right-justify them so that they take up the same amount of space, whether the coordinate has one, two, three, or four digits. Concatenating the right-justified string coordinates with '`X:` ' and '`Y:` ' labels gives us a neatly formatted string, which will be stored in `positionStr`.

At the end of your program, add the following code:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
--snip--
❶   print(positionStr, end='')
    print('\b' * len(positionStr), end='', flush=True)
```

This actually prints `positionStr` to the screen. The `end=''` keyword argument to `print()` prevents the default newline character from being added to the end of the printed line. It's possible to erase text you've already printed to the screen—but only for the most recent line of text. Once you print a newline character, you can't erase anything printed before it.

To erase text, print the `\b` backspace escape character. This special character erases a character at the end of the current line on the screen. The line at ❶ uses string replication to produce a string with as many `\b` characters as the length of the string stored in `positionStr`, which has the effect of erasing the `positionStr` string that was last printed.

For a technical reason beyond the scope of this book, always pass `flush=True` to `print()` calls that print `\b` backspace characters. Otherwise, the screen might not update the text as desired.

Since the `while` loop repeats so quickly, the user won't actually notice that you're deleting and reprinting the whole number on the screen. For example, if the x-coordinate is 563 and the mouse moves one pixel to the right, it will look like only the 3 in 563 is changed to a 4.

When you run the program, there will be only two lines printed. They should look like something like this:

```
Press Ctrl-C to quit.
X: 290 Y: 424
```

The first line displays the instruction to press `CTRL-C` to quit. The second line with the mouse coordinates will change as you move the mouse around the screen. Using this program, you'll be able to figure out the mouse coordinates for your GUI automation scripts.

Controlling Mouse Interaction

Now that you know how to move the mouse and figure out where it is on the screen, you're ready to start clicking, dragging, and scrolling.

Cicking the Mouse

To send a virtual mouse click to your computer, call the `pyautogui.click()` method. By default, this click uses the left mouse button and takes place wherever the mouse cursor is currently located. You can pass x- and y-coordinates of the click as optional first and second arguments if you want it to take place somewhere other than the mouse's current position.

If you want to specify which mouse button to use, include the `button` keyword argument, with a value of `'left'`, `'middle'`, or `'right'`. For example, `pyautogui.click(100, 150, button='left')` will click the left mouse button at the coordinates `(100, 150)`, while `pyautogui.click(200, 250, button='right')` will perform a right-click at `(200, 250)`.

Enter the following into the interactive shell:

```
>>> import pyautogui  
>>> pyautogui.click(10, 5)
```

You should see the mouse pointer move to near the top-left corner of your screen and click once. A full “click” is defined as pushing a mouse button down and then releasing it back up without moving the cursor. You can also perform a click by calling `pyautogui.mouseDown()`, which only pushes the mouse button down, and `pyautogui.mouseUp()`, which only releases the button. These functions have the same arguments as `click()`, and in fact, the `click()` function is just a convenient wrapper around these two function calls.

As a further convenience, the `pyautogui.doubleClick()` function will perform two clicks with the left mouse button, while the `pyautogui.rightClick()` and `pyautogui.middleClick()` functions will perform a click with the right and middle mouse buttons, respectively.

Dragging the Mouse

Dragging means moving the mouse while holding down one of the mouse buttons. For example, you can move files between folders by dragging the folder icons, or you can move appointments around in a calendar app.

PyAutoGUI provides the `pyautogui.dragTo()` and `pyautogui.dragRel()` functions to drag the mouse cursor to a new location or a location relative to its current one. The arguments for `dragTo()` and `dragRel()` are the same as `moveTo()` and `moveRel()`: the x-coordinate/horizontal movement, the y-coordinate/vertical movement, and an optional duration of time. (OS X does not drag correctly when the mouse moves too quickly, so passing a `duration` keyword argument is recommended.)

To try these functions, open a graphics-drawing application such as Paint on Windows, Paintbrush on OS X, or GNU Paint on Linux. (If you don't have a drawing application, you can use the online one at <http://sumopaint.com/>.) I will use PyAutoGUI to draw in these applications.

With the mouse cursor over the drawing application's canvas and the Pencil or Brush tool selected, enter the following into a new file editor window and save it as *spiralDraw.py*:

```
import pyautogui, time
❶ time.sleep(5)
❷ pyautogui.click()      # click to put drawing program in focus
distance = 200
while distance > 0:
❸     pyautogui.dragRel(distance, 0, duration=0.2)    # move right
❹     distance = distance - 5
❺     pyautogui.dragRel(0, distance, duration=0.2)    # move down
❻     pyautogui.dragRel(-distance, 0, duration=0.2)   # move left
     distance = distance - 5
     pyautogui.dragRel(0, -distance, duration=0.2)    # move up
```

When you run this program, there will be a five-second delay ❶ for you to move the mouse cursor over the drawing program's window with the Pencil or Brush tool selected. Then *spiralDraw.py* will take control of the mouse and click to put the drawing program in focus ❷. A window is in *focus* when it has an active blinking cursor, and the actions you take—like typing or, in this case, dragging the mouse—will affect that window. Once the drawing program is in focus, *spiralDraw.py* draws a square spiral pattern like the one in Figure 18-2.



Figure 18-2: The results from the `pyautogui.dragRel()` example

The distance variable starts at 200, so on the first iteration of the while loop, the first dragRel() call drags the cursor 200 pixels to the right, taking 0.2 seconds ❸. distance is then decreased to 195 ❹, and the second dragRel() call drags the cursor 195 pixels down ❺. The third dragRel() call drags the cursor –195 horizontally (195 to the left) ❻, distance is decreased to 190, and the last dragRel() call drags the cursor 190 pixels up. On each iteration, the mouse is dragged right, down, left, and up, and distance is slightly smaller than it was in the previous iteration. By looping over this code, you can move the mouse cursor to draw a square spiral.

You could draw this spiral by hand (or rather, by mouse), but you'd have to work slowly to be so precise. PyAutoGUI can do it in a few seconds!

NOTE

You could have your code draw the image using the pillow module's drawing functions—see Chapter 17 for more information. But using GUI automation allows you to make use of the advanced drawing tools that graphics programs can provide, such as gradients, different brushes, or the fill bucket.

Scrolling the Mouse

The final PyAutoGUI mouse function is scroll(), which you pass an integer argument for how many units you want to scroll the mouse up or down. The size of a unit varies for each operating system and application, so you'll have to experiment to see exactly how far it scrolls in your particular situation. The scrolling takes place at the mouse cursor's current position. Passing a positive integer scrolls up, and passing a negative integer scrolls down. Run the following in IDLE's interactive shell while the mouse cursor is over the IDLE window:

```
>>> pyautogui.scroll(200)
```

You'll see IDLE briefly scroll upward—and then go back down. The downward scrolling happens because IDLE automatically scrolls down to the bottom after executing an instruction. Enter this code instead:

```
>>> import pyperclip
>>> numbers = ''
>>> for i in range(200):
...     numbers = numbers + str(i) + '\n'
>>> pyperclip.copy(numbers)
```

This imports pyperclip and sets up an empty string, numbers. The code then loops through 200 numbers and adds each number to numbers, along with a newline. After pyperclip.copy(numbers), the clipboard will be loaded with 200 lines of numbers. Open a new file editor window and paste the text into it. This will give you a large text window to try scrolling in. Enter the following code into the interactive shell:

```
>>> import time, pyautogui
>>> time.sleep(5); pyautogui.scroll(100)
```

On the second line, you enter two commands separated by a semicolon, which tells Python to run the commands as if they were on separate lines. The only difference is that the interactive shell won't prompt you for input between the two instructions. This is important for this example because we want the call to `pyautogui.scroll()` to happen automatically after the wait. (Note that while putting two commands on one line can be useful in the interactive shell, you should still have each instruction on a separate line in your programs.)

After pressing ENTER to run the code, you will have five seconds to click the file editor window to put it in focus. Once the pause is over, the `pyautogui.scroll()` call will cause the file editor window to scroll up after the five-second delay.

Working with the Screen

Your GUI automation programs don't have to click and type blindly. PyAutoGUI has screenshot features that can create an image file based on the current contents of the screen. These functions can also return a Pillow `Image` object of the current screen's appearance. If you've been skipping around in this book, you'll want to read Chapter 17 and install the `pillow` module before continuing with this section.

On Linux computers, the `scrot` program needs to be installed to use the screenshot functions in PyAutoGUI. In a Terminal window, run `sudo apt-get install scrot` to install this program. If you're on Windows or OS X, skip this step and continue with the section.

Getting a Screenshot

To take screenshots in Python, call the `pyautogui.screenshot()` function. Enter the following into the interactive shell:

```
>>> import pyautogui  
>>> im = pyautogui.screenshot()
```

The `im` variable will contain the `Image` object of the screenshot. You can now call methods on the `Image` object in the `im` variable, just like any other `Image` object. Enter the following into the interactive shell:

```
>>> im.getpixel((0, 0))  
(176, 176, 175)  
>>> im.getpixel((50, 200))  
(130, 135, 144)
```

Pass `getpixel()` a tuple of coordinates, like `(0, 0)` or `(50, 200)`, and it'll tell you the color of the pixel at those coordinates in your image. The return value from `getpixel()` is an RGB tuple of three integers for the amount of red, green, and blue in the pixel. (There is no fourth value for alpha, because screenshot images are fully opaque.) This is how your programs can "see" what is currently on the screen.

Analyzing the Screenshot

Say that one of the steps in your GUI automation program is to click a gray button. Before calling the `click()` method, you could take a screenshot and look at the pixel where the script is about to click. If it's not the same gray as the gray button, then your program knows something is wrong. Maybe the window moved unexpectedly, or maybe a pop-up dialog has blocked the button. At this point, instead of continuing—and possibly wreaking havoc by clicking the wrong thing—your program can “see” that it isn’t clicking on the right thing and stop itself.

PyAutoGUI’s `pixelMatchesColor()` function will return `True` if the pixel at the given x- and y-coordinates on the screen matches the given color. The first and second arguments are integers for the x- and y-coordinates, and the third argument is a tuple of three integers for the RGB color the screen pixel must match. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> im = pyautogui.screenshot()
❶ >>> im.getpixel((50, 200))
(130, 135, 144)
❷ >>> pyautogui.pixelMatchesColor(50, 200, (130, 135, 144))
True
❸ >>> pyautogui.pixelMatchesColor(50, 200, (255, 135, 144))
False
```

After taking a screenshot and using `getpixel()` to get an RGB tuple for the color of a pixel at specific coordinates ❶, pass the same coordinates and RGB tuple to `pixelMatchesColor()` ❷, which should return `True`. Then change a value in the RGB tuple and call `pixelMatchesColor()` again for the same coordinates ❸. This should return `False`. This method can be useful to call whenever your GUI automation programs are about to call `click()`. Note that the color at the given coordinates must *exactly* match. If it is even slightly different—for example, `(255, 255, 254)` instead of `(255, 255, 255)`—then `pixelMatchesColor()` will return `False`.

Project: Extending the `mouseNow` Program

You could extend the `mouseNow.py` project from earlier in this chapter so that it not only gives the x- and y-coordinates of the mouse cursor’s current position but also gives the RGB color of the pixel under the cursor. Modify the code inside the `while` loop of `mouseNow.py` to look like this:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
--snip--
    positionStr = 'X: ' + str(x).rjust(4) + ' Y: ' + str(y).rjust(4)
    pixelColor = pyautogui.screenshot().getpixel((x, y))
    positionStr += ' RGB: (' + str(pixelColor[0]).rjust(3)
    positionStr += ', ' + str(pixelColor[1]).rjust(3)
    positionStr += ', ' + str(pixelColor[2]).rjust(3) + ')'
```

```
    print(positionStr, end=' ')
--snip--
```

Now, when you run *mouseNow.py*, the output will include the RGB color value of the pixel under the mouse cursor.

```
Press Ctrl-C to quit.
X: 406 Y: 17 RGB: (161, 50, 50)
```

This information, along with the `pixelMatchesColor()` function, should make it easy to add pixel color checks to your GUI automation scripts.

Image Recognition

But what if you do not know beforehand where PyAutoGUI should click? You can use image recognition instead. Give PyAutoGUI an image of what you want to click and let it figure out the coordinates.

For example, if you have previously taken a screenshot to capture the image of a Submit button in *submit.png*, the `locateOnScreen()` function will return the coordinates where that image is found. To see how `locateOnScreen()` works, try taking a screenshot of a small area on your screen; then save the image and enter the following into the interactive shell, replacing '*submit.png*' with the filename of your screenshot:

```
>>> import pyautogui
>>> pyautogui.locateOnScreen('submit.png')
(643, 745, 70, 29)
```

The four-integer tuple that `locateOnScreen()` returns has the x-coordinate of the left edge, the y-coordinate of the top edge, the width, and the height for the first place on the screen the image was found. If you're trying this on your computer with your own screenshot, your return value will be different from the one shown here.

If the image cannot be found on the screen, `locateOnScreen()` will return `None`. Note that the image on the screen must match the provided image perfectly in order to be recognized. If the image is even a pixel off, `locateOnScreen()` will return `None`.

If the image can be found in several places on the screen, `locateAllOnScreen()` will return a Generator object, which can be passed to `list()` to return a list of four-integer tuples. There will be one four-integer tuple for each location where the image is found on the screen. Continue the interactive shell example by entering the following (and replacing '*submit.png*' with your own image filename):

```
>>> list(pyautogui.locateAllOnScreen('submit.png'))
[(643, 745, 70, 29), (1007, 801, 70, 29)]
```

Each of the four-integer tuples represents an area on the screen. If your image is only found in one area, then using `list()` and `locateAllOnScreen()` just returns a list containing one tuple.

Once you have the four-integer tuple for the area on the screen where your image was found, you can click the center of this area by passing the tuple to the `center()` function to return x- and y-coordinates of the area's center. Enter the following into the interactive shell, replacing the arguments with your own filename, four-integer tuple, and coordinate pair:

```
>>> pyautogui.locateOnScreen('submit.png')
(643, 745, 70, 29)
>>> pyautogui.center((643, 745, 70, 29))
(678, 759)
>>> pyautogui.click((678, 759))
```

Once you have center coordinates from `center()`, passing the coordinates to `click()` should click the center of the area on the screen that matches the image you passed to `locateOnScreen()`.

Controlling the Keyboard

PyAutoGUI also has functions for sending virtual keypresses to your computer, which enables you to fill out forms or enter text into applications.

Sending a String from the Keyboard

The `pyautogui.typewrite()` function sends virtual keypresses to the computer. What these keypresses do depends on what window and text field have focus. You may want to first send a mouse click to the text field you want in order to ensure that it has focus.

As a simple example, let's use Python to automatically type the words *Hello world!* into a file editor window. First, open a new file editor window and position it in the upper-left corner of your screen so that PyAutoGUI will click in the right place to bring it into focus. Next, enter the following into the interactive shell:

```
>>> pyautogui.click(100, 100); pyautogui.typewrite('Hello world!')
```

Notice how placing two commands on the same line, separated by a semicolon, keeps the interactive shell from prompting you for input between running the two instructions. This prevents you from accidentally bringing a new window into focus between the `click()` and `typewrite()` calls, which would mess up the example.

Python will first send a virtual mouse click to the coordinates (100, 100), which should click the file editor window and put it in focus. The `typewrite()` call will send the text *Hello world!* to the window, making it look like Figure 18-3. You now have code that can type for you!



Figure 18-3: Using PyAutoGUI to click the file editor window and type Hello world! into it

By default, the `typewrite()` function will type the full string instantly. However, you can pass an optional second argument to add a short pause between each character. This second argument is an integer or float value of the number of seconds to pause. For example, `pyautogui.typewrite('Hello world!', 0.25)` will wait a quarter-second after typing *H*, another quarter-second after *e*, and so on. This gradual typewriter effect may be useful for slower applications that can't process keystrokes fast enough to keep up with PyAutoGUI.

For characters such as *A* or *!*, PyAutoGUI will automatically simulate holding down the SHIFT key as well.

Key Names

Not all keys are easy to represent with single text characters. For example, how do you represent SHIFT or the left arrow key as a single character? In PyAutoGUI, these keyboard keys are represented by short string values instead: 'esc' for the ESC key or 'enter' for the ENTER key.

Instead of a single string argument, a list of these keyboard key strings can be passed to `typewrite()`. For example, the following call presses the A key, then the B key, then the left arrow key twice, and finally the X and Y keys:

```
>>> pyautogui.typewrite(['a', 'b', 'left', 'left', 'X', 'Y'])
```

Because pressing the left arrow key moves the keyboard cursor, this will output XYab. Table 18-1 lists the PyAutoGUI keyboard key strings that you can pass to `typewrite()` to simulate pressing any combination of keys.

You can also examine the `pyautogui.KEYBOARD_KEYS` list to see all possible keyboard key strings that PyAutoGUI will accept. The 'shift' string refers to the left SHIFT key and is equivalent to 'shiftleft'. The same applies for 'ctrl', 'alt', and 'win' strings; they all refer to the left-side key.

Table 18-1: PyKeyboard Attributes

Keyboard key string	Meaning
'a', 'b', 'c', 'A', 'B', 'C', '1', '2', '3', '!','@','#', and so on	The keys for single characters
'enter' (or 'return' or '\n')	The ENTER key
'esc'	The ESC key
'shiftleft', 'shiftright'	The left and right SHIFT keys
'altright', 'altright'	The left and right ALT keys
'ctrlleft', 'ctrlright'	The left and right CTRL keys
'tab' (or '\t')	The TAB key
'backspace', 'delete'	The BACKSPACE and DELETE keys
'pageup', 'pagedown'	The PAGE UP and PAGE DOWN keys
'home', 'end'	The HOME and END keys
'up', 'down', 'left', 'right'	The up, down, left, and right arrow keys
'f1', 'f2', 'f3', and so on	The F1 to F12 keys
'volumemute', 'volumedown', 'volumeup'	The mute, volume down, and volume up keys (some keyboards do not have these keys, but your operating system will still be able to understand these simulated keypresses)
'pause'	The PAUSE key
'capslock', 'numlock', 'scrolllock'	The CAPS LOCK, NUM LOCK, and SCROLL LOCK keys
'insert'	The INS or INSERT key
'printscreen'	The PRTSC or PRINT SCREEN key
'winleft', 'winright'	The left and right WIN keys (on Windows)
'command'	The Command (⌘) key (on OS X)
'option'	The OPTION key (on OS X)

Pressing and Releasing the Keyboard

Much like the `mouseDown()` and `mouseUp()` functions, `pyautogui.keyDown()` and `pyautogui.keyUp()` will send virtual keypresses and releases to the computer. They are passed a keyboard key string (see Table 18-1) for their argument. For convenience, PyAutoGUI provides the `pyautogui.press()` function, which calls both of these functions to simulate a complete keypress.

Run the following code, which will type a dollar sign character (obtained by holding the SHIFT key and pressing 4):

```
>>> pyautogui.keyDown('shift'); pyautogui.press('4'); pyautogui.keyUp('shift')
```

This line presses down SHIFT, presses (and releases) 4, and then releases SHIFT. If you need to type a string into a text field, the `typewrite()` function is more suitable. But for applications that take single-key commands, the `press()` function is the simpler approach.

Hotkey Combinations

A *hotkey* or *shortcut* is a combination of keypresses to invoke some application function. The common hotkey for copying a selection is CTRL-C (on Windows and Linux) or ⌘-C (on OS X). The user presses and holds the CTRL key, then presses the C key, and then releases the C and CTRL keys. To do this with PyAutoGUI's `keyDown()` and `keyUp()` functions, you would have to enter the following:

```
pyautogui.keyDown('ctrl')
pyautogui.keyDown('c')
pyautogui.keyUp('c')
pyautogui.keyUp('ctrl')
```

This is rather complicated. Instead, use the `pyautogui.hotkey()` function, which takes multiple keyboard key string arguments, presses them in order, and releases them in the reverse order. For the CTRL-C example, the code would simply be as follows:

```
pyautogui.hotkey('ctrl', 'c')
```

This function is especially useful for larger hotkey combinations. In Word, the CTRL-ALT-SHIFT-S hotkey combination displays the Style pane. Instead of making eight different function calls (four `keyDown()` calls and four `keyUp()` calls), you can just call `hotkey('ctrl', 'alt', 'shift', 's')`.

With a new IDLE file editor window in the upper-left corner of your screen, enter the following into the interactive shell (in OS X, replace 'alt' with 'ctrl'):

```
>>> import pyautogui, time
>>> def commentAfterDelay():
❶    pyautogui.click(100, 100)
❷    pyautogui.typewrite('In IDLE, Alt-3 comments out a line.')
    time.sleep(2)
❸    pyautogui.hotkey('alt', '3')

>>> commentAfterDelay()
```

This defines a function `commentAfterDelay()` that, when called, will click the file editor window to bring it into focus ❶, type *In IDLE, Alt-3 comments out a line* ❷, pause for 2 seconds, and then simulate pressing the ALT-3 hotkey (or CTRL-3 on OS X) ❸. This keyboard shortcut adds two # characters to the current line, commenting it out. (This is a useful trick to know when writing your own code in IDLE.)

Review of the PyAutoGUI Functions

Since this chapter covered many different functions, here is a quick summary reference:

- `moveTo(x, y)` Moves the mouse cursor to the given `x` and `y` coordinates.
- `moveRel(xOffset, yOffset)` Moves the mouse cursor relative to its current position.
- `dragTo(x, y)` Moves the mouse cursor while the left button is held down.
- `dragRel(xOffset, yOffset)` Moves the mouse cursor relative to its current position while the left button is held down.
- `click(x, y, button)` Simulates a click (left button by default).
- `rightClick()` Simulates a right-button click.
- `middleClick()` Simulates a middle-button click.
- `doubleClick()` Simulates a double left-button click.
- `mouseDown(x, y, button)` Simulates pressing down the given button at the position `x, y`.
- `mouseUp(x, y, button)` Simulates releasing the given button at the position `x, y`.
- `scroll(units)` Simulates the scroll wheel. A positive argument scrolls up; a negative argument scrolls down.
- `typewrite(message)` Types the characters in the given message string.
- `typewrite([key1, key2, key3])` Types the given keyboard key strings.
- `press(key)` Presses the given keyboard key string.
- `keyDown(key)` Simulates pressing down the given keyboard key.
- `keyUp(key)` Simulates releasing the given keyboard key.
- `hotkey([key1, key2, key3])` Simulates pressing the given keyboard key strings down in order and then releasing them in reverse order.
- `screenshot()` Returns a screenshot as an `Image` object. (See Chapter 17 for information on `Image` objects.)

Project: Automatic Form Filler

Of all the boring tasks, filling out forms is the most dreaded of chores. It's only fitting that now, in the final chapter project, you will slay it. Say you have a huge amount of data in a spreadsheet, and you have to tediously retype it into some other application's form interface—with no intern to do it for you. Although some applications will have an Import feature that will allow you to upload a spreadsheet with the information, sometimes it seems that there is no other way than mindlessly clicking and typing for hours on end. You've come this far in this book; you know that *of course* there's another way.

The form for this project is a Google Docs form that you can find at <http://nostarch.com/automatestuff>. It looks like Figure 18-4.

The screenshot shows a Google Form window titled "Generic Form". The URL in the address bar is <https://docs.google.com/spreadsheet/viewform?fromEmail=1>. The form contains the following fields:

- Name ***: A text input field.
- Greatest Fear(s)**: A text input field.
- What is the source of your wizard powers?**: A dropdown menu set to "Wand".
- Robocop was the greatest action movie of the 1980s.**: A statement followed by a rating scale from 1 to 5. The scale has five points: 1, 2, 3, 4, 5. Below the scale are radio buttons labeled "Strongly Disagree" and "Strongly Agree".
- Additional Comments**: A large text area for comments.

At the bottom of the form is a "Submit" button and a note: "Never submit passwords through Google Forms."

Figure 18-4: The form used for this project

At a high level, here's what your program should do:

- Click the first text field of the form.
- Move through the form, typing information into each field.
- Click the Submit button.
- Repeat the process with the next set of data.

This means your code will need to do the following:

- Call `pyautogui.click()` to click the form and Submit button.
- Call `pyautogui.typewrite()` to enter text into the fields.
- Handle the `KeyboardInterrupt` exception so the user can press `CTRL-C` to quit.

Open a new file editor window and save it as `formFiller.py`.

Step 1: Figure Out the Steps

Before writing code, you need to figure out the exact keystrokes and mouse clicks that will fill out the form once. The *mouseNow.py* script on page 417 can help you figure out specific mouse coordinates. You need to know only the coordinates of the first text field. After clicking the first field, you can just press TAB to move focus to the next field. This will save you from having to figure out the x- and y-coordinates to click for every field.

Here are the steps for entering data into the form:

1. Click the Name field. (Use *mouseNow.py* to determine the coordinates after maximizing the browser window. On OS X, you may need to click twice: once to put the browser in focus and again to click the Name field.)
2. Type a name and then press TAB.
3. Type a greatest fear and then press TAB.
4. Press the down arrow key the correct number of times to select the wizard power source: once for *wand*, twice for *amulet*, three times for *crystal ball*, and four times for *money*. Then press TAB. (Note that on OS X, you will have to press the down arrow key one more time for each option. For some browsers, you may need to press the ENTER key as well.)
5. Press the right arrow key to select the answer to the RoboCop question. Press it once for *2*, twice for *3*, three times for *4*, or four times for *5*; or just press the spacebar to select *1* (which is highlighted by default). Then press TAB.
6. Type an additional comment and then press TAB.
7. Press the ENTER key to “click” the Submit button.
8. After submitting the form, the browser will take you to a page where you will need to click a link to return to the form page.

Note that if you run this program again later, you may have to update the mouse click coordinates, since the browser window might have changed position. To work around this, always make sure the browser window is maximized before finding the coordinates of the first form field. Also, different browsers on different operating systems might work slightly differently from the steps given here, so check that these keystroke combinations work for your computer before running your program.

Step 2: Set Up Coordinates

Load the example form you downloaded (Figure 18-4) in a browser and maximize your browser window. Open a new Terminal or command line window to run the *mouseNow.py* script, and then mouse over the Name field to figure out its the x- and y-coordinates. These numbers will be assigned to the `nameField` variable in your program. Also, find out the x- and y-coordinates and RGB tuple value of the blue Submit button. These values will be assigned to the `submitButton` and `submitButtonColor` variables, respectively.

Next, fill in some dummy data for the form and click **Submit**. You need to see what the next page looks like so that you can use *mouseNow.py* to find the coordinates of the *Submit another response* link on this new page.

Make your source code look like the following, being sure to replace all the values in italics with the coordinates you determined from your own tests:

```
#! python3
# formFiller.py - Automatically fills in the form.

import pyautogui, time

# Set these to the correct coordinates for your computer.
nameField = (648, 319)
submitButton = (651, 817)
submitButtonColor = (75, 141, 249)
submitAnotherLink = (760, 224)

# TODO: Give the user a chance to kill the script.

# TODO: Wait until the form page has loaded.

# TODO: Fill out the Name Field.

# TODO: Fill out the Greatest Fear(s) field.

# TODO: Fill out the Source of Wizard Powers field.

# TODO: Fill out the RoboCop field.

# TODO: Fill out the Additional Comments field.

# TODO: Click Submit.

# TODO: Wait until form page has loaded.

# TODO: Click the Submit another response link.
```

Now you need the data you actually want to enter into this form. In the real world, this data might come from a spreadsheet, a plaintext file, or a website, and it would require additional code to load into the program. But for this project, you'll just hardcode all this data in a variable. Add the following to your program:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

formData = [ {'name': 'Alice', 'fear': 'eavesdroppers', 'source': 'wand',
    'robocop': 4, 'comments': 'Tell Bob I said hi.'},
    {'name': 'Bob', 'fear': 'bees', 'source': 'amulet', 'robocop': 4,
    'comments': 'n/a'},
```

```
{'name': 'Carol', 'fear': 'puppets', 'source': 'crystal ball',
'robocop': 1, 'comments': 'Please take the puppets out of the
break room.'},
{'name': 'Alex Murphy', 'fear': 'ED-209', 'source': 'money',
'robocop': 5, 'comments': 'Protect the innocent. Serve the public
trust. Uphold the law.'},
]
```

--snip--

The `formData` list contains four dictionaries for four different names. Each dictionary has names of text fields as keys and responses as values. The last bit of setup is to set PyAutoGUI's `PAUSE` variable to wait half a second after each function call. Add the following to your program after the `formData` assignment statement:

```
pyautogui.PAUSE = 0.5
```

Step 3: Start Typing Data

A `for` loop will iterate over each of the dictionaries in the `formData` list, passing the values in the dictionary to the PyAutoGUI functions that will virtually type in the text fields.

Add the following code to your program:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

for person in formData:
    # Give the user a chance to kill the script.
    print('>>> 5 SECOND PAUSE TO LET USER PRESS CTRL-C <<<')
❶    time.sleep(5)

    # Wait until the form page has loaded.
❷    while not pyautogui.pixelMatchesColor(submitButton[0], submitButton[1],
        submitButtonColor):
        time.sleep(0.5)
```

--snip--

As a small safety feature, the script has a five-second pause ❶ that gives the user a chance to hit `CTRL-C` (or move the mouse cursor to the upper-left corner of the screen to raise the `FailSafeException` exception) to shut the program down in case it's doing something unexpected. Then the program waits until the Submit button's color is visible ❷, letting the program know that the form page has loaded. Remember that you figured out the

coordinate and color information in step 2 and stored it in the `submitButton` and `submitButtonColor` variables. To use `pixelMatchesColor()`, you pass the coordinates `submitButton[0]` and `submitButton[1]`, and the color `submitButtonColor`.

After the code that waits until the Submit button's color is visible, add the following:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

❶  print('Entering %s info...' % (person['name']))
❷  pyautogui.click(nameField[0], nameField[1])

    # Fill out the Name field.
❸  pyautogui.typewrite(person['name'] + '\t')

    # Fill out the Greatest Fear(s) field.
❹  pyautogui.typewrite(person['fear'] + '\t')

--snip--
```

We add an occasional `print()` call to display the program's status in its Terminal window to let the user know what's going on ❶.

Since the program knows that the form is loaded, it's time to call `click()` to click the Name field ❷ and `typewrite()` to enter the string in `person['name']` ❸. The '`\t`' character is added to the end of the string passed to `typewrite()` to simulate pressing TAB, which moves the keyboard focus to the next field, Greatest Fear(s). Another call to `typewrite()` will type the string in `person['fear']` into this field and then tab to the next field in the form ❹.

Step 4: Handle Select Lists and Radio Buttons

The drop-down menu for the “wizard powers” question and the radio buttons for the RoboCop field are trickier to handle than the text fields. To click these options with the mouse, you would have to figure out the x- and y-coordinates of each possible option. It's easier to use the keyboard arrow keys to make a selection instead.

Add the following to your program:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

    # Fill out the Source of Wizard Powers field.
❶  if person['source'] == 'wand':
❷      pyautogui.typewrite(['down', '\t'])
```

```

    elif person['source'] == 'amulet':
        pyautogui.typewrite(['down', 'down', '\t'])
    elif person['source'] == 'crystal ball':
        pyautogui.typewrite(['down', 'down', 'down', '\t'])
    elif person['source'] == 'money':
        pyautogui.typewrite(['down', 'down', 'down', 'down', '\t'])

    # Fill out the RoboCop field.
❸    if person['robocop'] == 1:
❹        pyautogui.typewrite([' ', '\t'])
    elif person['robocop'] == 2:
        pyautogui.typewrite(['right', '\t'])
    elif person['robocop'] == 3:
        pyautogui.typewrite(['right', 'right', '\t'])
    elif person['robocop'] == 4:
        pyautogui.typewrite(['right', 'right', 'right', '\t'])
    elif person['robocop'] == 5:
        pyautogui.typewrite(['right', 'right', 'right', 'right', '\t'])

--snip--

```

Once the drop-down menu has focus (remember that you wrote code to simulate pressing TAB after filling out the Greatest Fear(s) field), pressing the down arrow key will move to the next item in the selection list. Depending on the value in person['source'], your program should send a number of down arrow keypresses before tabbing to the next field. If the value at the 'source' key in this user's dictionary is 'wand' ❶, we simulate pressing the down arrow key once (to select *Wand*) and pressing TAB ❷. If the value at the 'source' key is 'amulet', we simulate pressing the down arrow key twice and pressing TAB, and so on for the other possible answers.

The radio buttons for the RoboCop question can be selected with the right arrow keys—or, if you want to select the first choice ❸, by just pressing the spacebar ❹.

Step 5: Submit the Form and Wait

You can fill out the Additional Comments field with the typewrite() function by passing person['comments'] as an argument. You can type an additional '\t' to move the keyboard focus to the next field or the Submit button. Once the Submit button is in focus, calling pyautogui.press('enter') will simulate pressing the ENTER key and submit the form. After submitting the form, your program will wait five seconds for the next page to load.

Once the new page has loaded, it will have a *Submit another response* link that will direct the browser to a new, empty form page. You stored the coordinates of this link as a tuple in submitAnotherLink in step 2, so pass these coordinates to pyautogui.click() to click this link.

With the new form ready to go, the script's outer for loop can continue to the next iteration and enter the next person's information into the form.

Complete your program by adding the following code:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

# Fill out the Additional Comments field.
pyautogui.typewrite(person['comments'] + '\t')

# Click Submit.
pyautogui.press('enter')

# Wait until form page has loaded.
print('Clicked Submit.')
time.sleep(5)

# Click the Submit another response link.
pyautogui.click(submitAnotherLink[0], submitAnotherLink[1])
```

Once the main for loop has finished, the program will have plugged in the information for each person. In this example, there are only four people to enter. But if you had 4,000 people, then writing a program to do this would save you a lot of time and typing!

Summary

GUI automation with the pyautogui module allows you to interact with applications on your computer by controlling the mouse and keyboard. While this approach is flexible enough to do anything that a human user can do, the downside is that these programs are fairly blind to what they are clicking or typing. When writing GUI automation programs, try to ensure that they will crash quickly if they're given bad instructions. Crashing is annoying, but it's much better than the program continuing in error.

You can move the mouse cursor around the screen and simulate mouse clicks, keystrokes, and keyboard shortcuts with PyAutoGUI. The pyautogui module can also check the colors on the screen, which can provide your GUI automation program with enough of an idea of the screen contents to know whether it has gotten offtrack. You can even give PyAutoGUI a screenshot and let it figure out the coordinates of the area you want to click.

You can combine all of these PyAutoGUI features to automate any mindlessly repetitive task on your computer. In fact, it can be downright hypnotic to watch the mouse cursor move on its own and see text appear on the screen automatically. Why not spend the time you saved by sitting back and watching your program do all your work for you? There's a certain satisfaction that comes from seeing how your cleverness has saved you from the boring stuff.

Practice Questions

1. How can you trigger PyAutoGUI’s fail safe to stop a program?
2. What function returns the current `resolution()`?
3. What function returns the coordinates for the mouse cursor’s current position?
4. What is the difference between `pyautogui.moveTo()` and `pyautogui.moveRel()`?
5. What functions can be used to drag the mouse?
6. What function call will type out the characters of "Hello world!"?
7. How can you do keypresses for special keys such as the keyboard’s left arrow key?
8. How can you save the current contents of the screen to an image file named `screenshot.png`?
9. What code would set a two second pause after every PyAutoGUI function call?

Practice Projects

For practice, write programs that do the following.

Looking Busy

Many instant messaging programs determine whether you are idle, or away from your computer, by detecting a lack of mouse movement over some period of time—say, ten minutes. Maybe you’d like to sneak away from your desk for a while but don’t want others to see your instant messenger status go into idle mode. Write a script to nudge your mouse cursor slightly every ten seconds. The nudge should be small enough so that it won’t get in the way if you do happen to need to use your computer while the script is running.

Instant Messenger Bot

Google Talk, Skype, Yahoo Messenger, AIM, and other instant messaging applications often use proprietary protocols that make it difficult for others to write Python modules that can interact with these programs. But even these proprietary protocols can’t stop you from writing a GUI automation tool.

The Google Talk application has a search bar that lets you enter a username on your friend list and open a messaging window when you press ENTER. The keyboard focus automatically moves to the new window. Other instant messenger applications have similar ways to open new message windows. Write a program that will automatically send out a notification message to a select group of people on your friend list. Your program may have to deal with exceptional cases, such as friends being offline, the chat window appearing at different coordinates on the screen, or confirmation

boxes that interrupt your messaging. Your program will have to take screenshots to guide its GUI interaction and adopt ways of detecting when its virtual keystrokes aren't being sent.

NOTE

You may want to set up some fake test accounts so that you don't accidentally spam your real friends while writing this program.

Game-Playing Bot Tutorial

There is a great tutorial titled “How to Build a Python Bot That Can Play Web Games” that you can find at <http://nostarch.com/automatestuff/>. This tutorial explains how to create a GUI automation program in Python that plays a Flash game called Sushi Go Round. The game involves clicking the correct ingredient buttons to fill customers’ sushi orders. The faster you fill orders without mistakes, the more points you get. This is a perfectly suited task for a GUI automation program—and a way to cheat to a high score! The tutorial covers many of the same topics that this chapter covers but also includes descriptions of PyAutoGUI’s basic image recognition features.

A

INSTALLING THIRD-PARTY MODULES



Beyond the standard library of modules packaged with Python, other developers have written their own modules to extend Python's capabilities even further. The primary way to install third-party modules is to use Python's pip tool. This tool securely downloads and installs Python modules onto your computer from <https://pypi.python.org/>, the website of the Python Software Foundation. PyPI, or the Python Package Index, is a sort of free app store for Python modules.

The pip Tool

The executable file for the pip tool is called *pip* on Windows and *pip3* on OS X and Linux. On Windows, you can find pip at *C:\Python34\Scripts\pip.exe*. On OS X, it is in */Library/Frameworks/Python.framework/Versions/3.4/bin/pip3*. On Linux, it is in */usr/bin/pip3*.

While pip comes automatically installed with Python 3.4 on Windows and OS X, you must install it separately on Linux. To install pip3 on Ubuntu or Debian Linux, open a new Terminal window and enter `sudo apt-get install python3-pip`. To install pip3 on Fedora Linux, enter `sudo yum install python3-pip` into a Terminal window. You will need to enter the administrator password for your computer in order to install this software.

Installing Third-Party Modules

The pip tool is meant to be run from the command line: You pass it the command `install` followed by the name of the module you want to install. For example, on Windows you would enter `pip install ModuleName`, where `ModuleName` is the name of the module. On OS X and Linux, you'll have to run pip3 with the `sudo` prefix to grant administrative privileges to install the module. You would need to type `sudo pip3 install ModuleName`.

If you already have the module installed but would like to upgrade it to the latest version available on PyPI, run `pip install -U ModuleName` (or `pip3 install -U ModuleName` on OS X and Linux).

After installing the module, you can test that it installed successfully by running `import ModuleName` in the interactive shell. If no error messages are displayed, you can assume the module was installed successfully.

You can install all of the modules covered in this book by running the commands listed next. (Remember to replace `pip` with `pip3` if you're on OS X or Linux.)

- `pip install send2trash`
- `pip install requests`
- `pip install beautifulsoup4`
- `pip install selenium`
- `pip install openpyxl`
- `pip install PyPDF2`
- `pip install python-docx` (install `python-docx`, not `docx`)
- `pip install imapclient`
- `pip install pyzmail`
- `pip install twilio`
- `pip install pillow`
- `pip install pyobjc-core` (on OS X only)
- `pip install pyobjc` (on OS X only)
- `pip install python3-xlib` (on Linux only)
- `pip install pyautogui`

NOTE

For OS X users: The `pyobjc` module can take 20 minutes or longer to install, so don't be alarmed if it takes a while. You should also install the `pyobjc-core` module first, which will reduce the overall installation time.

B

RUNNING PROGRAMS



If you have a program open in IDLE's file editor, running it is a simple matter of pressing F5 or selecting the Run ▶ Run Module menu item. This is an easy way to run programs while writing them, but opening IDLE to run your finished programs can be a burden. There are more convenient ways to execute Python scripts.

Shebang Line

The first line of all your Python programs should be a *shebang* line, which tells your computer that you want Python to execute this program. The shebang line begins with `#!`, but the rest depends on your operating system.

- On Windows, the shebang line is `#! python3`.
- On OS X, the shebang line is `#! /usr/bin/env python3`.
- On Linux, the shebang line is `#! /usr/bin/python3`.

You will be able to run Python scripts from IDLE without the shebang line, but the line is needed to run them from the command line.

Running Python Programs on Windows

On Windows, the Python 3.4 interpreter is located at *C:\Python34\python.exe*. Alternatively, the convenient *py.exe* program will read the shebang line at the top of the *.py* file's source code and run the appropriate version of Python for that script. The *py.exe* program will make sure to run the Python program with the correct version of Python if multiple versions are installed on your computer.

To make it convenient to run your Python program, create a *.bat batch file* for running the Python program with *py.exe*. To make a batch file, make a new text file containing a single line like the following:

```
@py.exe C:\path\to\your\pythonScript.py %*
```

Replace this path with the absolute path to your own program, and save this file with a *.bat* file extension (for example, *pythonScript.bat*). This batch file will keep you from having to type the full absolute path for the Python program every time you want to run it. I recommend you place all your batch and *.py* files in a single folder, such as *C:\MyPythonScripts* or *C:\Users\YourName\PythonScripts*.

The *C:\MyPythonScripts* folder should be added to the system path on Windows so that you can run the batch files in it from the Run dialog. To do this, modify the PATH environment variable. Click the **Start** button and type **Edit environment variables for your account**. This option should autocomplete after you've begun to type it. The Environment Variables window that appears will look like Figure B-1.

From System variables, select the Path variable and click **Edit**. In the Value text field, append a semicolon, type *C:\MyPythonScripts*, and then click **OK**. Now you can run any Python script in the *C:\MyPythonScripts* folder by simply pressing WIN-R and entering the script's name. Running *pythonScript*, for instance, will run *pythonScript.bat*, which in turn will save you from having to run the whole command *py.exe C:\MyPythonScripts\pythonScript.py* from the Run dialog.

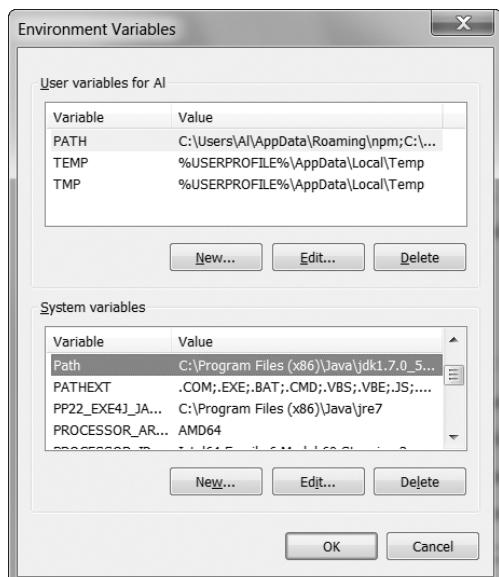


Figure B-1: The Environment Variables window on Windows

Running Python Programs on OS X and Linux

On OS X, selecting Applications ▶ Utilities ▶ Terminal will bring up a *Terminal* window. A Terminal window is a way to enter commands on your computer using only text, rather than clicking through a graphic interface. To bring up the Terminal window on Ubuntu Linux, press the WIN (or SUPER) key to bring up Dash and type in **Terminal**.

The Terminal window will begin in the home folder of your user account. If my username is *asweigart*, the home folder will be */Users/asweigart* on OS X and */home/asweigart* on Linux. The tilde (~) character is a shortcut for your home folder, so you can enter `cd ~` to change to your home folder. You can also use the `cd` command to change the current working directory to any other directory. On both OS X and Linux, the `pwd` command will print the current working directory.

To run your Python programs, save your *.py* file to your home folder. Then, change the *.py* file's permissions to make it executable by running `chmod +x pythonScript.py`. File permissions are beyond the scope of this book, but you will need to run this command on your Python file if you want to run the program from the Terminal window. Once you do so, you will be able to run your script whenever you want by opening a Terminal window and entering `./pythonScript.py`. The shebang line at the top of the script will tell the operating system where to locate the Python interpreter.

Running Python Programs with Assertions Disabled

You can disable the `assert` statements in your Python programs for a slight performance improvement. When running Python from the terminal, include the `-O` switch after `python` or `python3` and before the name of the *.py* file. This will run an optimized version of your program that skips the assertion checks.

C

ANSWERS TO THE PRACTICE QUESTIONS



This appendix contains the answers to the practice problems at the end of each chapter. I highly recommend that you take the time to work through these problems. Programming is more than memorizing syntax and a list of function names. As when learning a foreign language, the more practice you put into it, the more you will get out of it. There are many websites with practice programming problems as well. You can find a list of these at <http://nostarch.com/automatestuff/>.

Chapter 1

1. The operators are +, -, *, and /. The values are 'hello', -88.8, and 5.
2. The string is 'spam'; the variable is spam. Strings always start and end with quotes.
3. The three data types introduced in this chapter are integers, floating-point numbers, and strings.
4. An expression is a combination of values and operators. All expressions evaluate (that is, reduce) to a single value.
5. An expression evaluates to a single value. A statement does not.
6. The bacon variable is set to 20. The bacon + 1 expression does not reassign the value in bacon (that would need an assignment statement: bacon = bacon + 1).
7. Both expressions evaluate to the string 'spamspamspam'.
8. Variable names cannot begin with a number.
9. The int(), float(), and str() functions will evaluate to the integer, floating-point number, and string versions of the value passed to them.
10. The expression causes an error because 99 is an integer, and only strings can be concatenated to other strings with the + operator. The correct way is I have eaten ' + str(99) + ' burritos.'

Chapter 2

1. True and False, using capital T and F, with the rest of the word in lowercase
2. and, or, and not
3. True and True is True.
True and False is False.
False and True is False.
False and False is False.
True or True is True.
True or False is True.
False or True is True.
False or False is False.
not True is False.
not False is True.
4. False
False
True
False
False
True

5. ==, !=, <, >, <=, and >=.
 6. == is the equal to operator that compares two values and evaluates to a Boolean, while = is the assignment operator that stores a value in a variable.
 7. A condition is an expression used in a flow control statement that evaluates to a Boolean value.
 8. The three blocks are everything inside the if statement and the lines print('bacon') and print('ham').
-

```
print('eggs')
if spam > 5:
    print('bacon')
else:
    print('ham')
print('spam')
```

9. The code:
-

```
if spam == 1:
    print('Hello')
elif spam == 2:
    print('Howdy')
else:
    print('Greetings!')
```

10. Press CTRL-C to stop a program stuck in an infinite loop.
11. The break statement will move the execution outside and just after a loop.
The continue statement will move the execution to the start of the loop.
12. They all do the same thing. The range(10) call ranges from 0 up to (but not including) 10, range(0, 10) explicitly tells the loop to start at 0, and range(0, 10, 1) explicitly tells the loop to increase the variable by 1 on each iteration.

13. The code:
-

```
for i in range(1, 11):
    print(i)
```

and:

```
i = 1
while i <= 10:
    print(i)
    i = i + 1
```

14. This function can be called with spam.bacon().

Chapter 3

1. Functions reduce the need for duplicate code. This makes programs shorter, easier to read, and easier to update.
2. The code in a function executes when the function is called, not when the function is defined.
3. The `def` statement defines (that is, creates) a function.
4. A function consists of the `def` statement and the code in its `def` clause. A function call is what moves the program execution into the function, and the function call evaluates to the function's return value.
5. There is one global scope, and a local scope is created whenever a function is called.
6. When a function returns, the local scope is destroyed, and all the variables in it are forgotten.
7. A return value is the value that a function call evaluates to. Like any value, a return value can be used as part of an expression.
8. If there is no return statement for a function, its return value is `None`.
9. A `global` statement will force a variable in a function to refer to the global variable.
10. The data type of `None` is `NoneType`.
11. That `import` statement imports a module named `areallyourpetsnamederic`. (This isn't a real Python module, by the way.)
12. This function can be called with `spam.bacon()`.
13. Place the line of code that might cause an error in a `try` clause.
14. The code that could potentially cause an error goes in the `try` clause. The code that executes if an error happens goes in the `except` clause.

Chapter 4

1. The empty list value, which is a list value that contains no items. This is similar to how `''` is the empty string value.
2. `spam[2] = 'hello'` (Notice that the third value in a list is at index 2 because the first index is 0.)
3. `'d'` (Note that `'3' * 2` is the string `'33'`, which is passed to `int()` before being divided by `11`. This eventually evaluates to `3`. Expressions can be used wherever values are used.)
4. `'d'` (Negative indexes count from the end.)
5. `['a', 'b']`
6. `1`
7. `[3.14, 'cat', 11, 'cat', True, 99]`
8. `[3.14, 11, 'cat', True]`

9. The operator for list concatenation is +, while the operator for replication is *. (This is the same as for strings.)
10. While `append()` will add values only to the end of a list, `insert()` can add them anywhere in the list.
11. The `del` statement and the `remove()` list method are two ways to remove values from a list.
12. Both lists and strings can be passed to `len()`, have indexes and slices, be used in `for` loops, be concatenated or replicated, and be used with the `in` and `not in` operators.
13. Lists are mutable; they can have values added, removed, or changed. Tuples are immutable; they cannot be changed at all. Also, tuples are written using parentheses, (and), while lists use the square brackets, [and].
14. (42,) (The trailing comma is mandatory.)
15. The `tuple()` and `list()` functions, respectively
16. They contain references to list values.
17. The `copy.copy()` function will do a shallow copy of a list, while the `copy.deepcopy()` function will do a deep copy of a list. That is, only `copy.deepcopy()` will duplicate any lists inside the list.

Chapter 5

1. Two curly brackets: {}
2. {'foo': 42}
3. The items stored in a dictionary are unordered, while the items in a list are ordered.
4. You get a `KeyError` error.
5. There is no difference. The `in` operator checks whether a value exists as a key in the dictionary.
6. 'cat' in `spam` checks whether there is a 'cat' key in the dictionary, while 'cat' in `spam.values()` checks whether there is a value 'cat' for one of the keys in `spam`.
7. `spam.setdefault('color', 'black')`
8. `pprint.pprint()`

Chapter 6

1. Escape characters represent characters in string values that would otherwise be difficult or impossible to type into code.
2. \n is a newline; \t is a tab.
3. The \\ escape character will represent a backslash character.

4. The single quote in `Howl's` is fine because you've used double quotes to mark the beginning and end of the string.
5. Multiline strings allow you to use newlines in strings without the `\n` escape character.
6. The expressions evaluate to the following:
 - `'e'`
 - `'Hello'`
 - `'Hello'`
 - `'lo world!'`
7. The expressions evaluate to the following:
 - `'HELLO'`
 - `True`
 - `'hello'`
8. The expressions evaluate to the following:
 - `['Remember,', 'remember,', 'the', 'fifth', 'of', 'November.]`
 - `'There-can-be-only-one.'`
9. The `rjust()`, `ljust()`, and `center()` string methods, respectively
10. The `lstrip()` and `rstrip()` methods remove whitespace from the left and right ends of a string, respectively.

Chapter 7

1. The `re.compile()` function returns `Regex` objects.
2. Raw strings are used so that backslashes do not have to be escaped.
3. The `search()` method returns `Match` objects.
4. The `group()` method returns strings of the matched text.
5. Group 0 is the entire match, group 1 covers the first set of parentheses, and group 2 covers the second set of parentheses.
6. Periods and parentheses can be escaped with a backslash: `\.`, `\(`, and `\)`.
7. If the regex has no groups, a list of strings is returned. If the regex has groups, a list of tuples of strings is returned.
8. The `|` character signifies matching “either, or” between two groups.
9. The `?` character can either mean “match zero or one of the preceding group” or be used to signify nongreedy matching.
10. The `+` matches one or more. The `*` matches zero or more.
11. The `{3}` matches exactly three instances of the preceding group. The `{3,5}` matches between three and five instances.
12. The `\d`, `\w`, and `\s` shorthand character classes match a single digit, word, or space character, respectively.
13. The `\D`, `\W`, and `\S` shorthand character classes match a single character that is not a digit, word, or space character, respectively.

14. Passing `re.I` or `re.IGNORECASE` as the second argument to `re.compile()` will make the matching case insensitive.
15. The `.` character normally matches any character except the newline character. If `re.DOTALL` is passed as the second argument to `re.compile()`, then the dot will also match newline characters.
16. The `*` performs a greedy match, and the `*?` performs a nongreedy match.
17. Either `[0-9a-z]` or `[a-zA-Z]`
18. 'X drummers, X pipers, five rings, X hens'
19. The `re.VERBOSE` argument allows you to add whitespace and comments to the string passed to `re.compile()`.
20. `re.compile(r'^\d{1,3}(,\d{3})*$')` will create this regex, but other regex strings can produce a similar regular expression.
21. `re.compile(r'[A-Z][a-z]*\sNakamoto')`
22. `re.compile(r'(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs)\.', re.IGNORECASE)`

Chapter 8

1. Relative paths are relative to the current working directory.
2. Absolute paths start with the root folder, such as `/` or `C:\`.
3. The `os.getcwd()` function returns the current working directory. The `os.chdir()` function *changes* the current working directory.
4. The `.` folder is the current folder, and `..` is the parent folder.
5. `C:\bacon\eggs` is the dir name, while `spam.txt` is the base name.
6. The string `'r'` for read mode, `'w'` for write mode, and `'a'` for append mode
7. An existing file opened in write mode is erased and completely overwritten.
8. The `read()` method returns the file's entire contents as a single string value. The `readlines()` method returns a list of strings, where each string is a line from the file's contents.
9. A shelf value resembles a dictionary value; it has keys and values, along with `keys()` and `values()` methods that work similarly to the dictionary methods of the same names.

Chapter 9

1. The `shutil.copy()` function will copy a single file, while `shutil.copytree()` will copy an entire folder, along with all its contents.
2. The `shutil.move()` function is used for renaming files, as well as moving them.

3. The `send2trash` functions will move a file or folder to the recycle bin, while `shutil` functions will permanently delete files and folders.
4. The `zipfile.ZipFile()` function is equivalent to the `open()` function; the first argument is the filename, and the second argument is the mode to open the ZIP file in (read, write, or append).

Chapter 10

1. `assert(spam >= 10, 'The spam variable is less than 10.')`
2. `assert(eggs.lower() != bacon.lower(), 'The eggs and bacon variables are the same!') or assert(eggs.upper() != bacon.upper(), 'The eggs and bacon variables are the same!')`
3. `assert(False, 'This assertion always triggers.')`
4. To be able to call `logging.debug()`, you must have these two lines at the start of your program:

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s - %(message)s')
```

5. To be able to send logging messages to a file named `programLog.txt` with `logging.debug()`, you must have these two lines at the start of your program:

```
import logging
>>> logging.basicConfig(filename='programLog.txt', level=logging.DEBUG,
format=' %(asctime)s - %(levelname)s - %(message)s')
```

6. DEBUG, INFO, WARNING, ERROR, and CRITICAL
7. `logging.disable(logging.CRITICAL)`
8. You can disable logging messages without removing the logging function calls. You can selectively disable lower-level logging messages. You can create logging messages. Logging messages provides a timestamp.
9. The Step button will move the debugger into a function call. The Over button will quickly execute the function call without stepping into it. The Out button will quickly execute the rest of the code until it steps out of the function it currently is in.
10. After you click Go, the debugger will stop when it has reached the end of the program or a line with a breakpoint.
11. A breakpoint is a setting on a line of code that causes the debugger to pause when the program execution reaches the line.
12. To set a breakpoint in IDLE, right-click the line and select **Set Breakpoint** from the context menu.

Chapter 11

1. The `webbrowser` module has an `open()` method that will launch a web browser to a specific URL, and that's it. The `requests` module can download files and pages from the Web. The `BeautifulSoup` module parses HTML. Finally, the `selenium` module can launch and control a browser.
2. The `requests.get()` function returns a `Response` object, which has a `text` attribute that contains the downloaded content as a string.
3. The `raise_for_status()` method raises an exception if the download had problems and does nothing if the download succeeded.
4. The `status_code` attribute of the `Response` object contains the HTTP status code.
5. After opening the new file on your computer in '`wb`' "write binary" mode, use a `for` loop that iterates over the `Response` object's `iter_content()` method to write out chunks to the file. Here's an example:

```
saveFile = open('filename.html', 'wb')
for chunk in res.iter_content(100000):
    saveFile.write(chunk)
```

6. F12 brings up the developer tools in Chrome. Pressing `CTRL-SHIFT-C` (on Windows and Linux) or `⌘-OPTION-C` (on OS X) brings up the developer tools in Firefox.
7. Right-click the element in the page, and select **Inspect Element** from the menu.
8. `'#main'`
9. `'.highlight'`
10. `'div div'`
11. `'button[value="favorite"]'`
12. `spam.getText()`
13. `linkElem.attrs`
14. The `selenium` module is imported with `from selenium import webdriver`.
15. The `find_element_*` methods return the first matching element as a `WebElement` object. The `find_elements_*` methods return a list of all matching elements as `WebElement` objects.
16. The `click()` and `send_keys()` methods simulate mouse clicks and keyboard keys, respectively.
17. Calling the `submit()` method on any element within a form submits the form.
18. The `forward()`, `back()`, and `refresh()` `WebDriver` object methods simulate these browser buttons.

Chapter 12

1. The `openpyxl.load_workbook()` function returns a `Workbook` object.
2. The `get_sheet_names()` method returns a `Worksheet` object.
3. Call `wb.get_sheet_by_name('Sheet1')`.
4. Call `wb.get_active_sheet()`.
5. `sheet['C5'].value` or `sheet.cell(row=5, column=3).value`
6. `sheet['C5'] = 'Hello'` or `sheet.cell(row=5, column=3).value = 'Hello'`
7. `cell.row` and `cell.column`
8. They return the highest column and row with values in the sheet, respectively, as integer values.
9. `openpyxl.cell.column_index_from_string('M')`
10. `openpyxl.cell.get_column_letter(14)`
11. `sheet['A1':'F1']`
12. `wb.save('example.xlsx')`
13. A formula is set the same way as any value. Set the cell's `value` attribute to a string of the formula text. Remember that formulas begin with the `=` sign.
14. When calling `load_workbook()`, pass `True` for the `data_only` keyword argument.
15. `sheet.row_dimensions[5].height = 100`
16. `sheet.column_dimensions['C'].hidden = True`
17. OpenPyXL 2.0.5 does not load freeze panes, print titles, images, or charts.
18. Freeze panes are rows and columns that will always appear on the screen. They are useful for headers.
19. `openpyxl.charts.Reference()`, `openpyxl.charts.Series()`, `openpyxl.charts.BarChart()`, `chartObj.append(seriesObj)`, and `add_chart()`

Chapter 13

1. A `File` object returned from `open()`
2. Read-binary ('rb') for `PdfFileReader()` and write-binary ('wb') for `PdfFileWriter()`
3. Calling `getPage(4)` will return a `Page` object for page 5, since page 0 is the first page.
4. The `numPages` variable stores an integer of the number of pages in the `PdfFileReader` object.
5. Call `decrypt('swordfish')`.
6. The `rotateClockwise()` and `rotateCounterClockwise()` methods. The degrees to rotate is passed as an integer argument.

7. `docx.Document('demo.docx')`
8. A document contains multiple paragraphs. A paragraph begins on a new line and contains multiple runs. Runs are contiguous groups of characters within a paragraph.
9. Use `doc.paragraphs`.
10. A Run object has these variables (*not* a Paragraph).
11. `True` always makes the Run object bolded and `False` makes it always not bolded, no matter what the style's bold setting is. `None` will make the Run object just use the style's bold setting.
12. Call the `docx.Document()` function.
13. `doc.add_paragraph('Hello there!')`
14. The integers 0, 1, 2, 3, and 4

Chapter 14

1. In Excel, spreadsheets can have values of data types other than strings; cells can have different fonts, sizes, or color settings; cells can have varying widths and heights; adjacent cells can be merged; and you can embed images and charts.
2. You pass a `File` object, obtained from a call to `open()`.
3. File objects need to be opened in read-binary ('`rb`') for Reader objects and write-binary ('`wb`') for Writer objects.
4. The `writerow()` method
5. The `delimiter` argument changes the string used to separate cells in a row. The `lineterminator` argument changes the string used to separate rows.
6. `json.loads()`
7. `json.dumps()`

Chapter 15

1. A reference moment that many date and time programs use. The moment is January 1st, 1970, UTC.
2. `time.time()`
3. `time.sleep(5)`
4. It returns the closest integer to the argument passed. For example, `round(2.4)` returns 2.
5. A `datetime` object represents a specific moment in time. A `timedelta` object represents a duration of time.
6. `threadObj = threading.Thread(target=spam)`
7. `threadObj.start()`

8. Make sure that code running in one thread does not read or write the same variables as code running in another thread.
9. `subprocess.Popen('c:\\Windows\\\\System32\\\\calc.exe')`

Chapter 16

1. SMTP and IMAP, respectively
2. `smtplib.SMTP()`, `smtpObj.ehlo()`, `smptObj.starttls()`, and `smtpObj.login()`
3. `imapclient.IMAPClient()` and `imapObj.login()`
4. A list of strings of IMAP keywords, such as '`BEFORE <date>`', '`FROM <string>`', or '`SEEN`'
5. Assign the variable `imaplib._MAXLINE` a large integer value, such as `10000000`.
6. The `pyzmail` module reads downloaded emails.
7. You will need the Twilio account SID number, the authentication token number, and your Twilio phone number.

Chapter 17

1. An RGBA value is a tuple of 4 integers, each ranging from 0 to 255. The four integers correspond to the amount of red, green, blue, and alpha (transparency) in the color.
2. A function call to `ImageColor.getcolor('CornflowerBlue', 'RGBA')` will return `(100, 149, 237, 255)`, the RGBA value for that color.
3. A box tuple is a tuple value of four integers: the left edge x-coordinate, the top edge y-coordinate, the width, and the height, respectively.
4. `Image.open('zophie.png')`
5. `imageObj.size` is a tuple of two integers, the width and the height.
6. `imageObj.crop((0, 50, 50, 50))`. Notice that you are passing a box tuple to `crop()`, not four separate integer arguments.
7. Call the `imageObj.save('new_filename.png')` method of the `Image` object.
8. The `ImageDraw` module contains code to draw on images.
9. `ImageDraw` objects have shape-drawing methods such as `point()`, `line()`, or `rectangle()`. They are returned by passing the `Image` object to the `ImageDraw.Draw()` function.

Chapter 18

1. Move the mouse to the top-left corner of the screen, that is, the `(0, 0)` coordinates.
2. `pyautogui.size()` returns a tuple with two integers for the width and height of the screen.

3. `pyautogui.position()` returns a tuple with two integers for the x- and y-coordinates of the mouse cursor.
4. The `moveTo()` function moves the mouse to absolute coordinates on the screen, while the `moveRel()` function moves the mouse relative to the mouse's current position.
5. `pyautogui.dragTo()` and `pyautogui.dragRel()`
6. `pyautogui.typewrite('Hello world!')`
7. Either pass a list of keyboard key strings to `pyautogui.typewrite()` (such as `'left'`) or pass a single keyboard key string to `pyautogui.press()`.
8. `pyautogui.screenshot('screenshot.png')`
9. `pyautogui.PAUSE = 2`

INDEX

Symbols

= (assignment) operator, 18, 34
\ (backslash), 124, 151, 162, 174–175
 line continuation character, 93
^ (caret symbol), 162
 matching beginning of string,
 159–160
 negative character classes, 159
: (colon), 38, 45, 54, 82, 127
{ (curly brackets), 105, 162
 greedy vs. nongreedy matching,
 156–157
 matching specific repetitions
 with, 156
\$ (dollar sign), 159–160, 162
. (dot character), 160–162
 using in paths, 175–176
 wildcard matches, 160–162
" (double quotes), 124
** (exponent) operator, 15
== (equal to) operator, 33, 34
/ (forward slash), 174–175
 division operator, 15, 88
> (greater than) operator, 33
>= (greater than or equal to)
 operator, 33
(hash character), 126
// (integer division/floored quotient)
 operator, 15
< (less than) operator, 33
<= (less than or equal to) operator, 33
% (modulus/remainder) operator,
 15, 88
* (multiplication) operator, 15, 83, 88
!= (not equal to) operator, 33
() (parentheses), 96–97, 152–153
| (pipe character), 153–154, 164–165
+ (plus sign), 155–156, 162
 addition operator, 15, 17, 83, 88
? (question mark), 154–155, 162
' (single quote), 124

[] (square brackets), 80, 162
* (star), 162
 using with wildcard character, 161
 zero or more matches with, 155
- (subtraction) operator, 15, 88
''' (triple quotes), 125, 164
_ (underscore), 20

A

%A directive, 344
%a directive, 344
absolute paths, 175–179
abspath() function, 177
add (+) operator, 15, 17, 83, 88

B

\b backspace escape character, 419
%B directive, 344
%b directive, 344
back() method, 261
backslash (\), 124, 151, 162, 174–175
BarChart() function, 290
basename() function, 178
BCC search key, 370
Beautiful Soup, 245. *See also* bs4 module
BeautifulSoup objects, 245–246
BEFORE search key, 369
binary files, 180–181, 184–185
binary operators, 35–37
bitwise or operator, 164–165
blank strings, 17
blocking execution, 337
blocks of code, 37–38
BODY search key, 369
bold attribute, 311
Boolean data type
 binary operators, 35–36
 flow control and, 32–33
 in operator, 87
 not in operator, 87
 “truthy” and “falsey” values, 53
 using binary and comparison
 operators together, 36–37
box tuples, 390
breakpoints, debugging using, 229–231
break statements
 overview, 49–50
 using in for loop, 55
browser, opening using `webbrowser`
 module, 234–236
bs4 module
 creating object from HTML,
 245–246
 finding element with `select()`
 method, 246–247
 getting attribute, 248
 overview, 245
built-in functions, 57
bulleted list, creating in Wiki markup,
 139–141
 copying and pasting clipboard,
 139–140
joining modified lines, 141
 overview, 139
separating lines of text, 140

C

calling functions, 23
call stack, defined, 217
camelcase, 21
caret symbol (^), 162
 matching beginning of string,
 159–160
 negative character classes, 159
Cascading Style Sheets (CSS)
 matching with selenium
 module, 258
 selectors, 246–247
case sensitivity, 21, 163
CC search key, 370
Cell objects, 268–269
cells, in Excel spreadsheets, 266
 accessing Cell object by its name,
 268–269
 merging and unmerging, 286–287
 writing values to, 278–279
center() method, 133–134, 426
chaining method calls, 398
character classes, 158–159, 162
character styles, 310
charts, Excel, 288–290
`chdir()` function, 175
Chrome, developer tools in, 242–243
`clear()` method, 258
`click()` function, 420, 430, 431
clicking mouse, 420
`click()` method, 259
clipboard, using string from, 236
CMYK color model, 389
colon (:), 38, 45, 54, 82, 127
color values
 CMYK vs. RGB color models, 389
 RGBA values, 388–389
`column_index_from_string()` function, 270
columns, in Excel spreadsheets
 setting height and width of,
 285–286
 slicing `Worksheet` objects to get `Cell`
 objects in, 270–272
Comcast mail, 363, 367
comma-delimited items, 80
command line arguments, 235
`commentAfterDelay()` function, 429
comments
 multiline, 126
 overview, 23

comparison operators
 overview, 33–35
 using binary operators with, 36–37

`compile()` function, 151, 152, 164–165

compressed files
 backing up folder into, 209–212
 creating ZIP files, 205–206
 extracting ZIP files, 205
 overview, 203–204
 reading ZIP files, 204

computer screen
 coordinates of, 415
 resolution of, 416

concatenation
 of lists, 83
 string, 17–18

concurrency issues, 349

conditions, defined, 37

`continue` statements
 overview, 50–53
 using in `for` loop, 55

Coordinated Universal Time (UTC), 336

coordinates
 of computer screen, 415
 of an image, 389–390

`copy()` function, 100–101, 135, 198, 394

`copytree()` function, 198–199

countdown project, 357–358
 counting down, 357
 overview, 357
 playing sound file, 357–358

`cProfile.run()` function, 337

crashes, program, 14

`create_sheet()` method, 278

CRITICAL level, 224

cron, 354

cropping images, 393–394

CSS (Cascading Style Sheets)
 matching with `selenium` module, 258
 selectors, 246–247

CSV files
 defined, 319
 delimiter for, 324
 format overview, 320
 line terminator for, 324
 Reader objects, 321
 reading data in loop, 322
 removing header from, 324–327
 looping through CSV files, 325
 overview, 324–325
 reading in CSV file, 325–326
 writing out CSV file, 326–327

Writer objects, 322–323

curly brackets (`{}`), 105, 162
 greedy vs. nongreedy matching, 156–157
 matching specific repetitions with, 156

current working directory, 175

D

\D character class, 158

\d character class, 158

%d directive, 344

data structures
 algebraic chess notation, 112–113
 tic-tac-toe board, 113–117

data types
 Booleans, 32
 defined, 16
 dictionaries, 105–106
 floating-point numbers, 17
 integers, 17
 `list()` function, 97
 lists, 80
 mutable vs. immutable, 94–96
 `None` value, 65
 strings, 17
 `tuple()` function, 97
 tuples, 96–97

`datetime` module
 arithmetic using, 343
 converting objects to strings, 344–345
 converting strings to objects, 345
 `fromtimestamp()` function, 341
 `now()` function, 341
 overview, 341–342, 346
 pausing program until time, 344
 `timedelta` data type, 342–343
 `total_seconds()` method, 342

`datetime` objects, 341–342
 converting to strings, 344–345
 converting from strings to, 345

`debug()` function, 222

debugging
 assertions, 219–221
 defined, 4
 getting traceback as string, 217–218
 in IDLE
 overview, 225–227
 stepping through program, 227–229
 using breakpoints, 229–231

debugging (*continued*)
logging
 disabling, 224–225
 to file, 225
 levels of, 223–224
 logging module, 221–223
 print() function and, 223
 raising exceptions, 216–217
DEBUG level, 223
decimal numbers. *See* floating-point numbers
decode() method, 374–375
decryption, of PDF files, 297–298
deduplicating code, 62
deepcopy() function, 100–101
def statements, 62
 with parameters, 63
DELETED search key, 370
delete_messages() method, 375
deleting files/folders
 permanently, 200–201
 using send2trash module, 201–202
del statements, 84
dictionaries
 copy() function, 100–101
 deepcopy() function, 100–101
 get() method, 109
 in operator, 109
 items() method, 107–108
 keys() method, 107–108
 lists vs., 106–107
 nesting, 117–119
 not in operator, 109
 overview, 105–106
 setdefault() method, 110–111
 values() method, 107–108
directories
 absolute vs. relative paths, 175–176
 backslash vs. forward slash, 174–175
 copying, 198–199
 creating, 176
 current working directory, 175
 defined, 173–174
 deleting permanently, 200–201
 deleting using send2trash module, 201–202
 moving, 199–200
os.path module
 absolute paths in, 177–179
 file sizes, 179–180
 folder contents, 179–180
 overview, 177
path validity, 180
relative paths in, 177–179
renaming, 199–200
walking, 202–203
dirname() function, 178
disable() function, 224
division (/) operator, 15, 88
Document objects, 307–308
dollar sign (\$), 159–160, 162
dot character (.), 160–162
 using in paths, 175–176
wildcard matches, 160–162
dot-star character (.*), 161
doubleClick() function, 420, 430
double quotes ("), 124
double_strike attribute, 311
downloading
 files from web, 239–240
 web pages, 237–238
 XKCD comics, 251–256, 350–352
DRAFT search key, 370
dragging mouse, 420–422
dragRel() function, 420, 422, 430
dragTo() function, 420, 430
drawing on images
 ellipses, 407
 example program, 407–408
 ImageDraw module, 406
 lines, 406–407
 points, 406
 polygons, 407
 rectangles, 407
 text, 408–410
dumps() function, 329
duration keyword arguments, 416

E

ehlo() method, 364, 379
elements, HTML, 240
elif statements, 40–45
ellipse() method, 407
else statements, 39–40
email addresses, extracting, 165–169
 creating regex, 166–167
 finding matches on clipboard, 167–168
joining matches into a string, 168
 overview, 165–166
emails
 deleting, 375
 disconnecting from server, 375–376
fetching

folders, 368–369
getting message content, 372–373
logging into server, 368
overview, 366–367
raw messages, 373–375
`gmail_search()` method, 372
IMAP, 366
marking message as read, 372–373
searching, 368–371
sending
 connecting to SMTP server, 363–364
 disconnecting from server, 366
 logging into server, 364–365
 overview, 362
 reminder, 376–380
 sending “hello” message, 364
 sending message, 365
 TLS encryption, 364
SMTP, 362
`emboss` attribute, 311
encryption, of PDF files, 302–303
`endswith()` method, 131
epoch timestamps, 336, 341, 346
equal to (==) operator, 33, 34
ERROR level, 224
errors
 crashes and, 14
 help for, 8–9
escape characters, 124–125
evaluation, defined, 14
Excel spreadsheets
 application support, 265–266
 charts in, 288–290
 column width, 285–286
 converting between column letters
 and numbers, 270
 creating documents, 277
 creating worksheets, 278
 deleting worksheets, 278
 font styles, 282–284
 formulas in, 284–285
 freezing panes, 287–288
 getting cell values, 268–269
 getting rows and columns, 270–272
 getting worksheet names, 268
 merging and unmerging cells, 286–287
 opening documents, 267
 `openpyxl` module, 266
 overview, 266–267
reading files
 overview, 272–273
 populating data structure, 274–275
 reading data, 273–274
 writing results to file, 275–276
and reminder emails project, 376–380
row height, 285–286
saving workbooks, 277
updating, 279–281
 overview, 279–280
 setup, 280
workbooks vs., 266
writing values to cells, 278–279
Exception objects, 217
exceptions
 assertions and, 219–221
 getting traceback as string, 217–218
 handling, 72–74
 raising, 216–217
execution, program
 defined, 31
 overview, 38
 pausing until specific time, 344
 terminating program with
 `sys.exit()`, 58
`exists()` function, 180
exit codes, 353–354
`expand` keyword, 398
exponent (** operator, 15
expressions
 conditions and, 37
 in interactive shell, 14–16
`expunge()` method, 375
extensions, file, 173
`extractall()` method, 205
extracting ZIP files, 205
`extract()` method, 205

F

`FailSafeException` exception, 434
“falsey” values, 53
`fetch()` method, 371, 372–373
file editor, 21
file management
 absolute vs. relative paths, 175–176
 backslash vs. forward slash, 174–175
compressed files
 backing up to, 209–212
 creating ZIP files, 205–206

file management (*continued*)
 compressed files (*continued*)
 extracting ZIP files, 205
 overview, 203–204
 reading ZIP files, 204
 creating directories, 176
 current working directory, 175
 multiclipboard project, 191–193
 opening files, 181–182
 os.path module
 absolute paths in, 177–179
 file sizes, 179–180
 folder contents, 179–180
 overview, 177
 path validity, 180
 relative paths in, 177–179
 overview, 173–174
 paths, 173–174
 plaintext vs. binary files, 180–181
 reading files, 182–183
 renaming files, date styles, 206–209
 saving variables with `pformat()`
 function, 185–186
 send2trash module, 201–202
 shelve module, 184–185
 shutil module
 copying files/folders, 198–199
 deleting files/folders, 200–201
 moving files/folders, 199–200
 renaming files/folders, 199–200
 walking directory trees, 202–203
 writing files, 183–184
filenames, defined, 173
File objects, 182
`findall()` method, 157–158
`find_element_by_*` methods, 257–258
`find_elements_by_*` methods, 257–258
Firefox, developer tools in, 243
FLAGGED search key, 370
flipping images, 398–399
`float()` function, 25–28
floating-point numbers
 integer equivalence, 27
 overview, 17
 rounding, 338
flow control
 binary operators, 35–36
 blocks of code, 37–38
 Boolean values and, 32–33
 break statements, 49–50
 comparison operators, 33–35
 conditions, 37
continue statements, 50–53
elif statements, 40–45
else statements, 39–40
if statements, 38–39
overview, 31–32
using binary and comparison
 operators together, 36–37
while loops, 45–49
folders
 absolute vs. relative paths, 175–176
 backing up to ZIP file, 209–212
 creating new ZIP file, 211
 figuring out ZIP filename,
 210–211
 walking directory tree, 211–212
 backslash vs. forward slash, 174–175
 copying, 198–199
 creating, 176
 current working directory, 175
 defined, 173–174
 deleting permanently, 200–201
 deleting using `send2trash` module,
 201–202
 moving, 199–200
 os.path module
 absolute paths in, 177–179
 file sizes, 179–180
 folder contents, 179–180
 overview, 177
 path validity, 180
 relative paths in, 177–179
 renaming, 199–200
 walking directory trees, 202–203
Font objects, 282–283
font styles, in Excel spreadsheets,
 282–284
for loops
 overview, 53–56
 using dictionary items in, 108
 using lists with, 86
format attribute, 392
`format_description` attribute, 392
`formData` list, 434
form filler project, 430–437
 overview, 430–431
 radio buttons, 435–436
 select lists, 435–436
 setting up coordinates, 432–434
 steps in process, 431
 submitting form, 436–437
 typing data, 434–435

formulas, in Excel spreadsheets, 284–285
`forward()` method, 261
forward slash (/), 174–175
`FROM` search key, 370
`fromtimestamp()` function, 341, 346
functions. *See also names of individual functions*
 arguments, 23, 63
 as “black box”, 72
 built-in, 57
 `def` statements, 63
 exception handling, 72–74
 keyword arguments, 65–66
 `None` value and, 65
 overview, 61–62
 parameters, 63
 return values, 63–65

G

`get_active_sheet()` method, 268
`get_addresses()` method, 374
`get_attribute()` method, 258
`getcolor()` function, 388–389, 393
`get_column_letter()` function, 270
`getcwd()` function, 175
`get()` function
 overview, 109
 `requests` module, 237
`get_highest_column()` method, 269, 377
`get_highest_row()` method, 269
`get_payload()` method, 374–375
`getpixel()` function, 400, 423, 424
`get_sheet_by_name()` method, 268
`get_sheet_names()` method, 268
`getsize()` function, 179
`get_subject()` method, 374
`getText()` function, 308–309
GIF format, 392
global scope, 70–71
Gmail, 363, 365, 367
`gmail_search()` method, 372
Google Maps, 234–236
graphical user interface automation.
 See GUI (graphical user interface) automation
greater than (>) operator, 33
greater than or equal to (>=)
 operator, 33
greedy matching
 dot-star for, 161
 in regular expressions, 156–157

`group()` method, 151, 152–153
groups, regular expression
 matching
 greedy, 156–157
 nongreedy, 157
 one or more, 155–156
 optional, 154–155
 specific repetitions, 156
 zero or more, 155
 using parentheses, 152–153
 using pipe character in, 153–154

Guess the Number program, 74–76

GUI (graphical user interface)
 automation. *See also*
 form filler project
 controlling keyboard, 426–429
 hotkey combinations, 429
 key names, 427–428
 pressing and releasing, 428–429
 sending string from keyboard,
 426–427
 controlling mouse, 415–417,
 419–423
 clicking mouse, 420
 dragging mouse, 420–422
 scrolling mouse, 422–423
 determining mouse position,
 417–419
 image recognition, 425–426
 installing `pyautogui` module, 414
 logging out of program, 414
 overview, 413–414
 screenshots, 423–424
 stopping program, 414–415

H

`%H` directive, 344
hash character (#), 126
headings, Word document, 314–315
help
 asking online, 9–10
 for error messages, 8–9
hotkey combinations, 429
`hotkey()` function, 429, 430
Hotmail.com, 363, 367
HTML (Hypertext Markup Language)
 browser developer tools and,
 242–243
 finding elements, 244
 learning resources, 240
 overview, 240–241
 viewing page source, 241–242

I

- %I directive, 344
- id attribute, 241
- IDLE (interactive development environment)
 - creating programs, 21–22
 - debugging in
 - overview, 225–227
 - stepping through program, 227–229
 - using breakpoints, 229–231
 - expressions in, 14–16
 - overview, 8
 - running scripts outside of, 136
 - starting, 7–8
- if statements
 - overview, 38–39
 - using in while loop, 46–47
- imageDraw module, 406
- imageDraw objects, 406–408
- ImageFont objects, 408–410
- Image objects, 391–399
- images
 - adding logo to, 401–405
 - attributes for, 392–393
 - box tuples, 390
 - color values in, 388–389
 - coordinates in, 389–390
 - copying and pasting in, 394–396
 - cropping, 393–394
 - drawing on
 - example program, 407–408
 - ellipses, 407
 - ImageDraw module, 406
 - lines, 406–407
 - points, 406
 - polygons, 407
 - rectangles, 407
 - text, 408–410
 - flipping, 398–399
 - opening with Pillow, 390–391
 - pixel manipulation, 400
 - recognition of, 425–426
 - resizing, 397
 - RGBA values, 388–389
 - rotating, 398–399
 - transparent pixels, 397
- IMAP (Internet Message Access Protocol)
 - defined, 366
 - deleting messages, 375
- disconnecting from server, 375–376
- fetching messages, 372–375
- folders, 368–369
- logging into server, 368
- searching messages, 368–371
- imapclient module, 366
- IMAPClient objects, 367–368
- immutable data types, 94–96
- importing modules
 - overview, 57–58
- pyautogui module, 417
- imprint attribute, 311
- im variable, 423
- indentation, 93
- indexes
 - for dictionaries. *See* keys, dictionary for lists
 - changing values using, 83
 - getting value using, 80–81
 - negative, 82
 - removing values from list
 - using, 84
 - for strings, 126–127
- IndexError, 106
- index() method, 89
- infinite loops, 49, 51, 418
- INFO level, 223
- in operator
 - using with dictionaries, 109
 - using with lists, 87
 - using with strings, 127
- input() function
 - overview, 23–24, 89–90
 - using for sensitive information, 365
- installing
 - openpyxl module, 266
 - pyautogui module, 414
 - Python, 6–7
 - selenium module, 256
 - third-party modules, 441–442
- int, 17. *See also* integers
- integer division/floored quotient (/) operator, 15
- integers
 - floating-point equivalence, 27
 - overview, 17
- interactive development environment.
 - See* IDLE (interactive development environment)
- interactive shell. *See* IDLE
- Internet Explorer, developer tools in, 242–243

Internet Message Access Protocol. *See*
 IMAP (Internet Message
 Access Protocol)
interpreter, Python, 7
`int()` function, 25–28
`isabs()` function, 177
`isalnum()` method, 129–131
`isalpha()` method, 129–130
`isdecimal()` method, 129–131
`isdir()` function, 180
`is_displayed()` method, 258
`is_enabled()` method, 258
`.isfile()` function, 180
`islower()` method, 128–129
`is_selected()` method, 258
`isspace()` method, 130
`istitle()` method, 130
`isupper()` method, 128–129
italic attribute, 311
`items()` method, 107–108
`iter_content()` method, 239–240

J

`%j` directive, 344
`join()` method, 131–132, 174–175,
 177, 352
JPEG format, 392
JSON files
 APIs for, 327–328
 defined, 319–320
 format overview, 327–328
 reading, 328–329
 and weather data project, 329–332
 writing, 329
justifying text, 133–134

K

keyboard
 controlling, with PyAutoGUI
 hotkey combinations, 429
 pressing and releasing keys,
 428–429
 sending string from keyboard,
 426–427
 key names, 427–428
`KeyboardInterrupt` exception, 340,
 417, 418
`keyDown()` function, 428, 429, 430
keys, dictionary, 105

`keys()` method, 107–108
`keyUp()` function, 428, 429, 430
keyword arguments, 65–66

L

LARGER search key, 370
launchd, 354–355
launching programs
 and countdown project, 357–358
 opening files with default
 applications, 355–356
 opening websites, 355
 overview, 352–354
 passing command line arguments
 to processes, 354
`poll()` method, 353
running Python scripts, 355
scheduling, 354–355
`sleep()` function, 355
`wait()` method, 354
`len()` function, 307–308
 finding number of values in list, 83
 overview, 24–25
less than (`<`) operator, 33
less than or equal to (`<=`) operator, 33
LibreOffice, 265, 306
line breaks, Word document, 315
`LineChart()` function, 290
line continuation character (`\`), 93
`line()` method, 406–407
linked styles, 310
Linux
 backslash vs. forward slash, 174–175
 cron, 354
 installing Python, 7
 installing third-party modules, 442
 launching processes from
 Python, 353
logging out of automation
 program, 414
opening files with default
 applications, 355
`pip` tool on, 441–442
Python support, 4
running Python programs on, 445
starting IDLE, 8
 Unix philosophy, 356
`listdir()` function, 179
`list_folders()` method, 368–369
`list()` function, 321, 426

lists
append() method, 89–90
augmented assignment operators,
 88–89
changing values using index, 83
concatenation of, 83
copy() function, 100–101
deepcopy() function, 100–101
dictionaries vs., 106–107
finding number of values using
 len(), 83
getting sublists with slices, 82–83
getting value using index, 80–81
index() method, 89
in operator, 87
insert() method, 89–90
list() function, 97
Magic 8 Ball example program
 using, 92–93
multiple assignment trick, 87–88
mutable vs. immutable data types,
 94–96
negative indexes, 82
nesting, 117–119
not in operator, 87
overview, 80
remove() method, 90–91
removing values from, 84
replication of, 83
sort() method, 91–92
storing variables as, 84–85
 using with for loops, 86
ljust() method, 133–134
load_workbook() function, 267
loads() function, 328–329, 331
local scope, 67–70
locateAllOnScreen() function, 426
locateOnScreen() function, 425
location attribute, 258
logging
 disabling, 224–225
 to file, 225
 levels of, 223–224
 print() function and, 223
logging module, 221–223
logging out, of automation
 program, 414
login() method, 364, 368, 379
logo, adding to an image, 401–406
 looping over files, 402–403
 opening logo image, 401–402
 overview, 404
 resizing image, 403–404

logout() method, 375–376
LogRecord objects, 221
loops
 break statements, 49–50
 continue statements, 50–53
 for loop, 53–56
 range() function for, 56–57
 reading data from CSV file, 322
 using lists with, 86
 while loop, 45–49
lower() method, 128–129
lstrip() method, 134–135

M

%M directive, 344
%m directive, 344
Mac OS X. *See* OS X
Magic 8 Ball example program, 92–93
makedirs() function, 176, 403
maps, open when location is copied,
 234–236
 figuring out URL, 234–235
 handling clipboard content, 236
 handling command line argument,
 235–236
 launching browser, 236
 overview, 234
Match objects, 151
math
 operators for, 15
 programming and, 4
mergePage() method, 302
Message objects, 381–382
methods
 chaining calls, 398
 defined, 89
 dictionary
 get() method, 109
 items() method, 107–108
 keys() method, 107–108
 setdefault() method, 110–111
 values() method, 107–108
list
 append() method, 89–90
 index() method, 89
 insert() method, 89–90
 remove() method, 90–91
 sort() method, 91–92
string
 center() method, 133–134
 copy() method, 135
 endswith() method, 131

`isalnum()` method, 129–131
`isalpha()` method, 129–130
`isdecimal()` method, 129–131
`islower()` method, 128–129
`isspace()` method, 130
`istitle()` method, 130
`isupper()` method, 128–129
`join()` method, 131–132
`ljust()` method, 133–134
`lower()` method, 128–129
`lstrip()` method, 134–135
`paste()` method, 135
`rjust()` method, 133–134
`rstrip()` method, 134–135
`split()` method, 131–133
`startswith()` method, 131
`strip()` method, 134–135
`upper()` method, 128–129

Microsoft Windows. *See* Windows OS

`middleClick()` function, 420, 430

modules

- importing, 57–58
- third-party, installing, 442

modulus/remainder (%) operator, 15, 88

Monty Python, 4

mouse

- controlling, 415–417, 419–423
 - clicking mouse, 420
 - dragging mouse, 420–422
 - scrolling mouse, 422–423
- determining position of, 417–419
- locating, 417–419
 - getting coordinates, 418–419
 - handling `KeyboardInterrupt` exception, 418
 - importing `pyautogui` module, 418
 - infinite loop, 418
 - overview, 417
- and pixels, identifying colors of, 424–425

`mouseDown()` function, 420, 430

`mouse.position()` function, 418

`mouseUp()` function, 430

`move()` function, 199–200

`moveRel()` function, 416, 417, 420, 430

`moveTo()` function, 416, 420, 430

moving files/folders, 199–200

multiclipboard project, 191–193

- listing keywords, 193
- loading keyword content, 193
- overview, 191

saving clipboard content, 192

setting up shelf file, 192

multiline comments, 126

multiline strings, 125–126

multiple assignment trick, 87–88

multiplication (*) operator, 15, 83, 88

multithreading

- concurrency issues, 349
- downloading multiple images, , 350–352
 - creating and starting threads, 351–352
 - using `downloadXkcd()` function, 350–351
 - waiting for threads to end, 352
- `join()` method, 352
- overview, 347–348
- passing arguments to threads, 348–349
- `start()` method, 348, 349
- `Thread()` function, 347–348

mutable data types, 94–96

N

`NameError`, 84

`namelist()` method, 204

negative character classes, 159

negative indexes, 82

nested lists and dictionaries, 117–119

newline keyword argument, 322

`None` value, 65

nongreedy matching

- dot, star, and question mark for, 161
- in regular expressions, 157

`not equal to (!=)` operator, 33

`not in` operator

- using with dictionaries, 109
- using with lists, 87
- using with strings, 127

`not` operator, 36

NOT search key, 370

`now()` function, 341, 346

O

ON search key, 369

`open()` function, 181–182, 234, 355–356, 391

opening files, 181–182

OpenOffice, 265, 306

open program, 355

- openpyxl module, installing, 266
- operators
- augmented assignment, 88–89
 - binary, 35–36
 - comparison, 33–35
 - defined, 14
 - math, 15
 - using binary and comparison operators together, 36–37
- order of operations, 15
- or operator, 36
- OR search key, 370
- OS X
- backslash vs. forward slash, 174–175
 - installing Python, 7
 - installing third-party modules, 442
 - launchd, 354
 - launching processes from Python, 356
 - logging out of automation program, 414
 - opening files with default applications, 355–356
 - pip tool on, 441–442
 - Python support, 4
 - running Python programs on, 445
 - starting IDLE, 8
 - Unix philosophy, 356
- outline attribute, 311
- Outlook.com, 363, 367
- P**
- %p directive, 344
- page breaks, Word document, 315
- Page objects, 297
- Paragraph objects, 307
- paragraphs, Word document, 309–310
- parameters, function, 63
- parentheses (), 96–97, 152–153
- parsing, defined, 245
- passing arguments, 23
- passing references, 100
- passwords
- application-specific, 365
 - managing project, 136–138
 - command-line arguments, 137
 - copying password, 137–138
 - data structures, 136–137
 - overview, 136
- pastebin.com, 10
- paste() method, 135, 394, 395
- paths
- absolute vs. relative, 175–176
 - backslash vs. forward slash, 174–175
 - current working directory, 175
 - overview, 173–174
- os.path module
- absolute paths in, 177–179
 - file sizes, 179–180
 - folder contents, 179–180
 - overview, 177
 - path validity, 180
 - relative paths in, 177–179
- PAUSE variable, 415, 434
- PdfFileReader objects, 297–298
- PDF files
- combining pages from multiple files, 303–306
 - adding pages, 303
 - finding PDF files, 304
 - opening PDFs, 304–305
 - overview, 303
 - saving results, 305–306
- creating, 298–299
- decrypting, 297–298
- encrypting, 302–303
- extracting text from, 296–297
- format overview, 295–296
- pages in
- copying, 299–300
 - overlaying, 301–302
 - rotating, 300–301
- PdfWriter objects, 298–299
- printf() function
- overview, 111–112
 - saving variables in text files using, 185–186
- phone numbers, extracting, 165–169
- creating regex, 166
 - finding matches on clipboard, 167–168
 - joining matches into a string, 168
 - overview, 165–166
- Pillow
- copying and pasting in images, 394–396
 - cropping images, 393–394
 - drawing on images
 - ellipses, 407
 - example program, 407–408
 - ImageDraw module, 406
 - lines, 406–407

points, 406
polygons, 407
rectangles, 407
text, 408–410
flipping images, 398–399
image attributes, 392–393
module, 388
opening images, 390–391
pixel manipulation, 400
resizing images, 397
rotating images, 398–399
transparent pixels, 397
pipe character (|), 153–154, 164–165
pip tool, 441–442
`pixelMatchesColor()` function, 424, 435
pixels, 388, 400
plaintext files, 180–181
plus sign (+), 155–156, 162
PNG format, 392
`point()` method, 406
`poll()` method, 353
`polygon()` method, 407
`Popen()` function, 352–353
 opening files with default
 applications, 355–356
 passing command line
 arguments to, 354
`position()` function, 417, 419
`pprint()` function, 111–112
precedence of math operators, 15
`press()` function, 429, 430, 436
`print()` function, 435
 logging and, 223
 overview, 23
 passing multiple arguments to, 66
 using variables with, 24
processes
 and countdown project, 357–358
 defined, 352
 opening files with default
 applications, 355–356
 opening websites, 355
 passing command line
 arguments to, 354
`poll()` method, 353
`Popen()` function, 352–353
`wait()` method, 354
profiling code, 336–337
programming
 blocks of code, 37–38
 comments, 23
 creativity needed for, 5
deduplicating code, 62
defined, 3
exception handling, 72–74
execution, program, 38
functions as “black boxes”, 72
global scope, 70–71
indentation, 93
local scope, 67–70
math and, 4
Python, 4
terminating program with
 `sys.exit()`, 58

projects

- Adding Bullets to Wiki Markup, 139–141
- Adding a Logo, 401–406
- Automatic Form Filler, 430–437
- Backing Up a Folder into a ZIP File, 209–212
- Combining Select Pages from Many PDFs, 303–306
- Downloading All XKCD Comics, 251–256
- Extending the mouseNow Program, 424–425
- Fetching Current Weather Data, 329–332
- Generating Random Quiz Files, 186–191
- “I’m Feeling Lucky” Google Search, 248–251
- “Just Text Me” Module, 383
- `mapIt.py` with the `webbrowser` Module, 234–236
- Multiclipboard, 191–193
- Multithreaded XKCD Downloader, 350–352
- Password Locker, 136–138
- Phone Number and Email Address Extractor, 165–169
- Reading Data from a Spreadsheet, 272–276
- Removing the Header from CSV Files, 324–327
- Renaming Files with American-Style Dates to European-Style Dates, 206–209
- Sending Member Dues Reminder Emails, 376–379
- Simple Countdown Program, 357–358
- Super Stopwatch, 338–340

- projects (*continued*)
 - Updating a Spreadsheet, 279–281
 - “Where Is the Mouse Right Now?”, 417–419
 - `putpixel()` method, 400
 - `pyautogui.click()` function, 431
 - `pyautogui.click()` method, 420
 - `pyautogui.doubleClick()` function, 420
 - `pyautogui.dragTo()` function, 420
 - `pyautogui.FailSafeException` exception, 415
 - `pyautogui.hotkey()` function, 429
 - `pyautogui.keyDown()` function, 428
 - `pyautogui.keyUp()` function, 428
 - `pyautogui.middleClick()` function, 420
 - `pyautogui` module
 - form filler project, 430–437
 - controlling keyboard, 426–429
 - hotkey combinations, 429
 - key names, 427–428
 - pressing and releasing keys, 428–429
 - sending string from keyboard, 426–427
 - controlling mouse, 415–417, 419–423
 - clicking mouse, 420
 - dragging mouse, 420–422
 - scrolling mouse, 422–423
 - documentation for, 414
 - fail-safe feature, 415
 - functions, 430
 - image recognition, 425–426
 - importing, 417
 - installing, 414
 - pausing function calls, 415
 - screenshots, 423–424
 - `pyautogui.mouseDown()` function, 420
 - `pyautogui.moveRel()` function, 416, 417
 - `pyautogui.moveTo()` function, 416
 - `pyautogui.PAUSE` variable, 415
 - `pyautogui.position()` function, 419
 - `pyautogui.press()` function, 436
 - `pyautogui.rightClick()` function, 420
 - `pyautogui.screenshot()` function, 423
 - `pyautogui.size()` function, 416
 - `pyautogui.typewrite()` function, 426, 427, 431
 - `py.exe` program, 444–445
 - `pyobjc` module, 442
- PyPDF2 module**
 - combining pages from multiple PDFs, 303–306
 - creating PDFs, 298–299
 - decrypting PDFs, 297–298
 - encrypting PDFs, 302–303
 - extracting text from PDFs, 296–297
 - format overview, 295–296
 - pages in PDFs
 - copying, 299–300
 - overlaying, 301–302
 - rotating, 300–301
- pyperclip module**, 135
- Python**
 - data types, 16–17
 - downloading, 6
 - example program, 21–22
 - help, 8–9
 - installing, 6–7
 - interactive shell, 8
 - interpreter, defined, 7
 - math and, 4
 - overview, 4
 - programming overview, 3
 - starting IDLE, 7–8
- python-docx module**, 306
- pyzmail module**, 366, 373–375
- PyzMessage objects**, 373–375

Q

- question mark (?), 154–155, 162
- `quit()` method, 261, 366, 379
- quiz generator, 186–190
 - creating quiz file, 188
 - creating answer options, 189
 - overview, 186
 - shuffling question order, 188
 - storing quiz data in dictionary, 187
 - writing content to files, 189–191

R

- radio buttons, 435–436
- `raise_for_status()` method, 238
- raise keyword, 216–217
- `range()` function, 56–57
- raw strings, 125, 151
- Reader objects, 321–322
- reading files, 182–183, 204
- `readlines()` method, 182

`read()` method, 182
`rectangle()` method, 407
Reddit, 9
Reference objects, 289
references
 overview, 97–99
 passing, 100
`refresh()` method, 261
Regex objects
 creating, 150
 matching, 151
regular expressions
 beginning of string matches, 159–160
 case sensitivity, 163
 character classes, 158–159
 creating Regex objects, 150–151
 defined, 147–148
 end of string matches, 159–160
 extracting phone numbers and emails addresses, 165–169
 `findall()` method, 157–158
 finding text without, 148–150
 greedy matching, 156–157
 grouping
 matching specific repetitions, 156
 one or more matches, 155–156
 optional matching, 154–155
 using parentheses, 152–153
 using pipe character in, 153–154
 zero or more matches, 155
 HTML and, 243
 matching with, 151–152
 multiple arguments for `compile()` function, 164–165
 nongreedy matching, 157
 patterns for, 150
 spreading over multiple lines, 164
 substituting strings using, 163–164
 symbol reference, 162
 wildcard character, 160–162
relative paths, 175–179
`relpath()` function, 177, 178
remainder/modulus (%) operator, 15, 88
`remove()` method, 90–91
`remove_sheet()` method, 278
renaming files/folders, 199–200
 date styles, 206–209
 creating regex for dates, 206
 identifying dates in filenames, 207–208
overview, 206
renaming files, 209
replication
 of lists, 83
 string, 18
requests module
 downloading files, 239–240
 downloading pages, 237–238
resolution of computer screen, 416
Response objects, 237–238
return values, function, 63–65
reverse keyword, 91
RGBA values, 388–389
RGB color model, 389
`rightClick()` function, 420, 430
`rjust()` method, 133–134, 419
`rmdir()` function, 200
`rmtree()` function, 200
`rotateClockwise()` method, 300
`rotateCounterClockwise()` method, 300
rotating images, 398–399
rounding numbers, 338
rows, in Excel spreadsheets
 setting height and width of, 285–286
 slicing `Worksheet` objects to get Cell objects in, 270–272
`rstrip()` method, 134–135
rtl attribute, 311
Run objects, 310, 311–312
running programs
 on Linux, 445
 on OS X, 445
 overview, 443
 on Windows, 444–445
shebang line, 443–444

S

\s character class, 158
\s character class, 158
\\$ directive, 344
Safari, developer tools in, 243
`save()` method, 391
scope
 global, 70–71
 local, 67–70
`screenshot()` function, 423, 430
screenshots
 analyzing, 424
 getting, 423
scripts
 running from Python program, 355
 running outside of IDLE, 136

`scroll()` function, 422, 423, 430
scrolling mouse, 422–423
searching
 email, 368–371
 the Web, 248–251
 finding results, 249–250
 getting command line arguments, 249
 opening web browser for results, 250–251
 overview, 248
 requesting search page, 249
`search()` method, 151
`SEEN` search key, 370
see program, 355
`select_folder()` method, 369
`select lists`, 435–436
`select()` method, `bs4` module, 246–247
selectors, CSS, 246–247, 258
`selenium` module
 clicking buttons, 261
 finding elements, 257–259
 following links, 259
 installing, 256
 sending special keystrokes, 260–261
 submitting forms, 259–260
 using Firefox with, 256–257
`send2trash` module, 201–202
sending reminder emails, 376–379
 finding unpaid members, 378
 opening Excel file, 376–377
 overview, 376
 sending emails, 378–379
`send_keys()` method, 259–260
`sendmail()` method, 365, 379
sequence numbers, 373
sequences, 86
`setdefault()` method, 110–111
shadow attribute, 311
shebang line, 443–444
`shelve` module, 184–185
Short Message Service (SMS)
 sending messages, 381–382
 Twilio service, 380
`shutil` module
 deleting files/folders, 200–201
 moving files/folders, 199–200
 renaming files/folders, 199–200
`SID` (string ID), 382
Simple Mail Transfer Protocol. *See*
 `SMTP` (Simple Mail Transfer Protocol)
`SINCE` search key, 369
single quote ('), 124
single-threaded programs, 347
`size()` function, 416
`sleep()` function, 337–338, 344, 346, 355
slices
 getting sublists with, 82–83
 for strings, 126–127
`small_caps` attribute, 311
`SMALLER` search key, 370
SMS (Short Message Service)
 sending messages, 381–382
 Twilio service, 380
SMTP (Simple Mail Transfer Protocol)
 connecting to server, 363–364
 defined, 362
 disconnecting from server, 366
 logging into server, 364–365
 sending “hello” message, 364
 sending message, 365
 TLS encryption, 364
`SMTP` objects, 363–364
`sort()` method, 91–92
sound files, playing, 357–358
source code, defined, 3
`split()` method, 131–133, 178, 320
spreadsheets. *See* Excel spreadsheets
square brackets [], 80
Stack Overflow, 9
standard library, 57
star (*), 161, 162
 using with wildcard character, 161
 zero or more matches with, 155
`start()` method, 348, 349, 351
start program, 355
`startswith()` method, 131
`starttls()` method, 364, 379
step argument, 56
stopwatch project, 338–340
 overview, 338–339
 set up, 339
 tracking lap times, 339–340
`strftime()` function, 344–345, 346
`str()` function, 25–28, 97, 419
strike attribute, 311
string ID (SID), 382
strings
 `center()` method, 133–134
 concatenation, 17–18
 converting `datetime` objects to, 344–345
 converting to `datetime` objects, 345

copying and pasting, 135
double quotes for, 124
`endswith()` method, 131
escape characters, 124–125
extracting PDF text as, 296–297
getting traceback as, 217–218
indexes for, 126–127
`in` operator, 127
`isalnum()` method, 129–131
`isalpha()` method, 129–130
`isdecimal()` method, 129–131
`islower()` method, 128–129
`isspace()` method, 130
`istitle()` method, 130
`isupper()` method, 128–129
`join()` method, 131–132
literals, 124
`ljust()` method, 133–134
`lower()` method, 128–129
`lstrip()` method, 134–135
multiline, 125–126
mutable vs. immutable data types, 94–96
`not in` operator, 127
overview, 17
raw, 125
replication of, 18
`rjust()` method, 133–134
`rstrip()` method, 134–135
slicing, 126–127
`split()` method, 131–133
`startswith()` method, 131
`strip()` method, 134–135
substituting using regular expressions, 163–164
`upper()` method, 128–129
`strip()` method, 134–135
`strptime()` function, 345, 346
strs, 17, *See also* strings
Style objects, 282–283
SUBJECT search key, 369
sublists, getting with slices, 82–83
`sub()` method, 163–164
`submitButtonColor` variable, 432, 435
`submitButton` variable, 432
`submit()` method, 260
`subprocess` module, 335, 352–354
subtraction (-) operator, 15, 88
subtractive color model, 389
Sudoku puzzles, 4
`sys.exit()` function, 58

T

`tag_name` attribute, 258
Tag objects, 246–247
tags, HTML, 240
Task Scheduler, 354
termination, program, 22, 58
`text` attribute, 308, 311
text messaging
 automatic notifications, 383
 sending messages, 381–382
 Twilio service, 380
`text()` method, 408–410
TEXT search key, 369
`textsize()` method, 409
third-party modules, installing, 441–442
`Thread()` function, 347–348, 351
threading module, 335, 347
Thread objects, 347–348
threads
 concurrency issues, 349
 `join()` method, 352
 multithreading, 347–348
 image downloader, 350–352
 passing arguments to, 348–349
 processes vs., 352
tic-tac-toe board, 113–117
`timedelta` data type, 342–343, 346
`timedelta` objects, 342–343
time module
 overview, 346
 `sleep()` function, 337–338, 344
 stopwatch project, 338–340
 `time()` function, 336–337
TLS encryption, 364
top-level domains, 167
TO search key, 370
`total_seconds()` method, 342, 346
traceback, getting from error, 217–218
transparency, 388, 397
`transpose()` method, 399
triple quotes (''), 125, 164
`truestype()` function, 409
truth tables, 35–36
“truthy” values, 53
tuple data type
 overview, 96–97
 `tuple()` function, 97
`twilio` module, 380
`TwilioRestClient` objects, 381

Twilio service

- automatic text messages, 383
 - overview, 380
 - sending text messages, 381–382
- `TypeError`, 81, 94
- `typewrite()` function, 426, 427, 430, 431, 435, 436

Ubuntu

- Ubuntu, 7
 - cron, 354–355
 - launching processes from Python, 353
 - opening files with default applications, 355
 - Unix philosophy, 356
- `UNANSWERED` search key, 370
- `UNDELETED` search key, 370
- `underline` attribute, 311
- `underscore` (`_`), 20
- `UNDRAFT` search key, 370
- `UNFLAGGED` search key, 370
- Unicode encodings, 239
- Unix epoch, 336, 341, 346
- Unix philosophy, 356
- `unlink()` function, 200
- `UNSEEN` search key, 370
- `upper()` method, 128–129
- UTC (Coordinated Universal Time), 336

V

- `ValueError`, 88, 345
- values, defined, 14, 150
- `values()` method, 107–108
- variables. *See also* lists
 - assignment statements, 18–19
 - defined, 18
 - global, 70–71
 - initializing, 19
 - local, 67–70
 - naming, 20–21
 - `None` value and, 65
 - overwriting, 19–20
 - references, 97–99
 - saving with `shelve` module, 184–185
 - storing as list, 84–85
- Verizon mail, 363, 367
- volumes, defined, 174

W

- `\w` character class, 158
- `\w` character class, 158
- `%w` directive, 344
- `walk()` function, 202–203, 354
- `WARNING` level, 223
- weather data, fetching, 329–332
 - downloading JSON data, 330–331
 - getting location, 330
 - loading JSON data, 331–332
 - overview, 329
- `webbrowser` module
 - `open()` function, 355
 - opening browser using, 234–236
- `WebDriver` objects, 257
- `WebElement` objects, 257–258
- web scraping
 - `bs4` module
 - creating object from HTML, 245–246
 - finding element with `select()` method, 246–247
 - getting attribute, 248
 - overview, 245
 - downloading
 - files, 239–240
 - images, 251–256
 - pages, 237–238
 - and Google maps project, 234–236
 - and Google search project, 248–251
- HTML
 - browser developer tools and, 242–243
 - finding elements, 244
 - learning resources, 240
 - overview, 240–241
 - viewing page source, 241–242
- overview, 233–234
- `requests` module, 237–238
- `selenium` module
 - clicking buttons, 261
 - finding elements, 257–259
 - following links, 259
 - installing, 256
 - sending special keystrokes, 260–261
 - submitting forms, 259–260
 - using Firefox with, 256–257
- websites, opening from script, 355

while loops
 getting and printing mouse coordinates using, 418
 infinite, 418
 overview, 45–49
whitespace, removing, 134–135
wildcard character (.), 160–162
Windows OS
 backslash vs. forward slash, 174–175
 installing Python, 6–7
 installing third-party modules, 442
 launching processes from Python, 353
 logging out of automation program, 414
 opening files with default applications, 355–356
 pip tool on, 441–442
 Python support, 4
 running Python programs on, 444–445
 starting IDLE, 7
 Task Scheduler, 354
Word documents
 adding headings, 314–315
 creating documents with nondefault styles, 310–311
 format overview, 306–307
 getting text from, 308–309
 line/page breaks, 315
 pictures in, 315–316
 python-docx module, 306
 reading, 307–308
 Run object attributes, 311–312
 styling paragraphs, 309–310
 writing to file, 312–314
Workbook objects, 267
workbooks, Excel, 266
 creating worksheets, 278
 deleting worksheets, 278
 opening, 267
 saving, 277
Worksheet objects, 268
write() method, 183–184
Writer objects, 322–323
writerow() method, 323

X

XKCD comics
 downloading project, 251–256
 designing program, 252–253
 downloading web page, 253–254
 overview, 251–252
 saving image, 255–256
multithreaded downloading project, 350–352
 creating and starting threads, 351–352
 using `downloadXkcd()` function, 350–351
 waiting for threads to end, 352

Y

%Y directive, 344
%y directive, 344
Yahoo! Mail, 363, 367

Z

zipfile module
 creating ZIP files, 205–206
 extracting ZIP files, 205
 and folders, 209–212
 overview, 203–204
 reading ZIP files, 204
ZipFile objects, 204–205
ZipInfo objects, 204

RESOURCES

Visit <http://nostarch.com/automatestuff/> for resources, errata, and more information.

More no-nonsense books from



NO STARCH PRESS



PYTHON PLAYGROUND

**Geeky Weekend Projects
for the Curious Programmer**

by MAHESH VENKITACHALAM
MAY 2015, 304 pp., \$29.95
ISBN 978-1-59327-604-1



PYTHON CRASH COURSE

**A Hands-On, Project-Based
Introduction to Programming**

by ERIC MATTHES
JULY 2015, 624 pp., \$34.95
ISBN 978-1-59327-603-4



THE LINUX COMMAND LINE

A Complete Introduction

by WILLIAM E. SHOTTS, JR.
JANUARY 2012, 480 pp., \$39.95
ISBN 978-1-59327-389-7



JAVASCRIPT FOR KIDS

A Playful Introduction to Programming

by NICK MORGAN
DECEMBER 2014, 336 pp., \$34.95
ISBN 978-1-59327-408-5
full color



STATISTICS DONE WRONG

The Woefully Complete Guide

by ALEX REINHART
MARCH 2015, 176 pp., \$24.95
ISBN 978-1-59327-620-1



DATA VISUALIZATION WITH JAVASCRIPT

by STEPHEN A. THOMAS
MARCH 2015, 384 pp., \$39.95
ISBN 978-1-59327-605-8
full color

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM
WEB:
WWW.NOSTARCH.COM

**LEARN PYTHON.
GET STUFF DONE.**



If you've ever spent hours renaming files or updating hundreds of spreadsheet cells, you know how tedious tasks like these can be. But what if you could have your computer do them for you?

In *Automate the Boring Stuff with Python*, you'll learn how to use Python to write programs that do in minutes what would take you hours to do by hand—no prior programming experience required. Once you've mastered the basics of programming, you'll create Python programs that effortlessly perform useful and impressive feats of automation to:

- Search for text in a file or across multiple files
- Create, update, move, and rename files and folders
- Search the Web and download online content
- Update and format data in Excel spreadsheets of any size
- Split, merge, watermark, and encrypt PDFs

- Send reminder emails and text notifications
- Fill out online forms

Step-by-step instructions walk you through each program, and practice projects at the end of each chapter challenge you to improve those programs and use your newfound skills to automate similar tasks.

Don't spend your time doing work a well-trained monkey could do. Even if you've never written a line of code, you can make your computer do the grunt work. Learn how in *Automate the Boring Stuff with Python*.

ABOUT THE AUTHOR

Al Sweigart is a software developer and teaches programming to kids and adults. He has written several Python books for beginners, including *Hacking Secret Ciphers with Python*, *Invent Your Own Computer Games with Python*, and *Making Games with Python & Pygame*.

COVERS PYTHON 3



THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com

"I LIE FLAT."

This book uses a durable binding that won't snap shut.



ISBN: 978-1-59327-599-0

A standard linear barcode representing the ISBN number 978-1-59327-599-0.

9 781593 275990

\$29.95 (\$34.95 CDN)



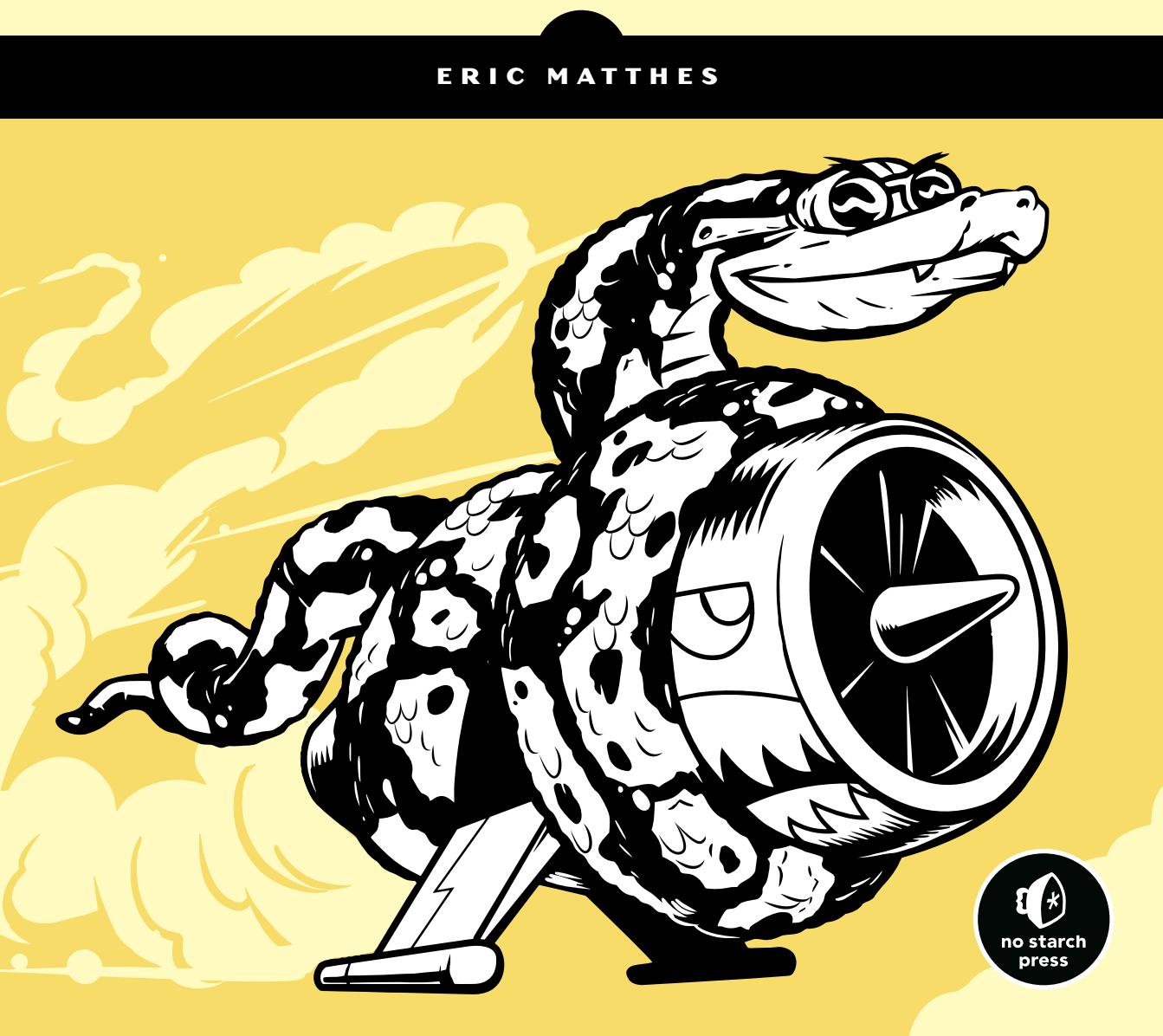
SHELF LIFE:
PROGRAMMING LANGUAGES /
PYTHON

2ND EDITION

PYTHON CRASH COURSE

A HANDS-ON, PROJECT-BASED
INTRODUCTION TO PROGRAMMING

ERIC MATTHES



PRAISE FOR PYTHON CRASH COURSE

“It has been interesting to see No Starch Press producing future classics that should be alongside the more traditional programming books. *Python Crash Course* is one of those books.”

—GREG LADEN, SCIENCEBLOGS

“Deals with some rather complex projects and lays them out in a consistent, logical, and pleasant manner that draws the reader into the subject.”

—FULL CIRCLE MAGAZINE

“Well presented with good explanations of the code snippets. The book works with you, one small step at a time, building more complex code, explaining what’s going on all the way.”

—FLICKTHROUGH REVIEWS

“Learning Python with *Python Crash Course* was an extremely positive experience! A great choice if you’re new to Python.”

—MIKKE GOES CODING

“Does what it says on the tin, and does it really well. . . . Presents a large number of useful exercises as well as three challenging and entertaining projects.”

—REALPYTHON.COM

“A fast-paced but comprehensive introduction to programming with Python, *Python Crash Course* is another superb book to add to your library and help you finally master Python.”

—TUTORIALEDGE.NET

“A brilliant option for complete beginners without any coding experience. If you’re looking for a solid, uncomplicated intro to this very deep language, I have to recommend this book.”

—WHATPIXEL.COM

“Contains literally everything you need to know about Python and even more.”

—FIREBEARSTUDIO.COM

PYTHON

CRASH COURSE

2ND EDITION

**A Hands-On, Project-Based
Introduction to Programming**

by Eric Matthes



San Francisco

PYTHON CRASH COURSE, 2ND EDITION. Copyright © 2019 by Eric Matthes.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-10: 1-59327-928-0

ISBN-13: 978-1-59327-928-8

Publisher: William Pollock

Production Editor: Riley Hoffman

Cover Illustration: Josh Ellingson

Cover and Interior Design: Octopod Studios

Developmental Editor: Liz Chadwick

Technical Reviewer: Kenneth Love

Copyeditor: Anne Marie Walker

Compositors: Riley Hoffman and Happenstance Type-O-Rama

Proofreader: James Fraleigh

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.

245 8th Street, San Francisco, CA 94103

phone: 1.415.863.9900; info@nostarch.com

www.nostarch.com

The Library of Congress has catalogued the first edition as follows:

Matthes, Eric, 1972-

Python crash course : a hands-on, project-based introduction to programming / by Eric Matthes.

pages cm

Includes index.

Summary: "A project-based introduction to programming in Python, with exercises. Covers general programming concepts, Python fundamentals, and problem solving. Includes three projects - how to create a simple video game, use data visualization techniques to make graphs and charts, and build an interactive web application"-- Provided by publisher.

ISBN 978-1-59327-603-4 -- ISBN 1-59327-603-6

1. Python (Computer program language) I. Title.

QA76.73.P98M38 2015

005.13'3--dc23

2015018135

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

About the Author

Eric Matthes is a high school science and math teacher living in Alaska, where he teaches an introductory Python course. He has been writing programs since he was five years old. Eric currently focuses on writing software that addresses inefficiencies in education and brings the benefits of open source software to the field of education. In his spare time he enjoys climbing mountains and spending time with his family.

About the Technical Reviewer

Kenneth Love has been a Python programmer, teacher, and conference organizer for many years. He has spoken and taught at many conferences, been a Python and Django freelancer, and is currently a software engineer for O'Reilly Media. Kenneth is co-creator of the django-braces package, which provides several handy mixins for Django's class-based views. You can keep up with him on Twitter at @kennethlove.

For my father, who always made time to
answer my questions about programming,
and for Ever, who is just beginning to ask me
his questions

BRIEF CONTENTS

Preface to the Second Edition	xxvii
Acknowledgments	xxxii
Introduction	xxxiii

PART I: BASICS 1

Chapter 1: Getting Started	3
Chapter 2: Variables and Simple Data Types	15
Chapter 3: Introducing Lists	33
Chapter 4: Working with Lists	49
Chapter 5: if Statements	71
Chapter 6: Dictionaries	91
Chapter 7: User Input and while Loops	113
Chapter 8: Functions	129
Chapter 9: Classes	157
Chapter 10: Files and Exceptions	183
Chapter 11: Testing Your Code	209

PART II: PROJECTS 223

Project 1: Alien Invasion

Chapter 12: A Ship that Fires Bullets	227
Chapter 13: Aliens!	255
Chapter 14: Scoring	279

Project 2: Data Visualization

Chapter 15: Generating Data	305
Chapter 16: Downloading Data	333
Chapter 17: Working with APIs	359

Project 3: Web Applications

Chapter 18: Getting Started with Django	379
Chapter 19: User Accounts	409
Chapter 20: Styling and Deploying an App	437
Afterword	465
Appendix A: Installation and Troubleshooting	467
Appendix B: Text Editors and IDEs	473
Appendix C: Getting Help	479
Appendix D: Using Git for Version Control	485
Index	495

CONTENTS IN DETAIL

PREFACE TO THE SECOND EDITION	xxvii
ACKNOWLEDGMENTS	xxxii
INTRODUCTION	xxxiii
Who Is This Book For?	xxxiv
What Can You Expect to Learn?	xxxiv
Online Resources	xxxv
Why Python?	xxxvi

PART I: BASICS **1**

1	GETTING STARTED	3
Setting Up Your Programming Environment	3	
Python Versions	4	
Running Snippets of Python Code	4	
About the Sublime Text Editor	4	
Python on Different Operating Systems	5	
Python on Windows	5	
Python on macOS	7	
Python on Linux	8	
Running a Hello World Program	9	
Configuring Sublime Text to Use the Correct Python Version	9	
Running <code>hello_world.py</code>	10	
Troubleshooting	11	
Running Python Programs from a Terminal	12	
On Windows	12	
On macOS and Linux	12	
Exercise 1-1: <code>python.org</code>	13	
Exercise 1-2: <code>Hello World</code> Typos	13	
Exercise 1-3: Infinite Skills	13	
Summary	13	
2	VARIABLES AND SIMPLE DATA TYPES	15
What Really Happens When You Run <code>hello_world.py</code>	15	
Variables	16	
Naming and Using Variables	17	
Avoiding Name Errors When Using Variables	17	
Variables Are Labels	18	
Exercise 2-1: Simple Message	19	
Exercise 2-2: Simple Messages	19	

Strings	19
Changing Case in a String with Methods	20
Using Variables in Strings	21
Adding Whitespace to Strings with Tabs or Newlines	22
Stripping Whitespace	22
Avoiding Syntax Errors with Strings	24
<i>Exercise 2-3: Personal Message</i>	25
<i>Exercise 2-4: Name Cases</i>	25
<i>Exercise 2-5: Famous Quote</i>	25
<i>Exercise 2-6: Famous Quote 2</i>	25
<i>Exercise 2-7: Stripping Names</i>	25
Numbers	25
Integers	26
Floats	26
Integers and Floats	27
Underscores in Numbers	28
Multiple Assignment	28
Constants	28
<i>Exercise 2-8: Number Eight</i>	29
<i>Exercise 2-9: Favorite Number</i>	29
Comments	29
How Do You Write Comments?	29
What Kind of Comments Should You Write?	29
<i>Exercise 2-10: Adding Comments</i>	30
The Zen of Python	30
<i>Exercise 2-11: Zen of Python</i>	31
Summary	32

3 INTRODUCING LISTS 33

What Is a List?	33
Accessing Elements in a List	34
Index Positions Start at 0, Not 1	35
Using Individual Values from a List	35
<i>Exercise 3-1: Names</i>	36
<i>Exercise 3-2: Greetings</i>	36
<i>Exercise 3-3: Your Own List</i>	36
Changing, Adding, and Removing Elements	36
Modifying Elements in a List	36
Adding Elements to a List	37
Removing Elements from a List	38
<i>Exercise 3-4: Guest List</i>	42
<i>Exercise 3-5: Changing Guest List</i>	42
<i>Exercise 3-6: More Guests</i>	42
<i>Exercise 3-7: Shrinking Guest List</i>	43
Organizing a List	43
Sorting a List Permanently with the <code>sort()</code> Method	43
Sorting a List Temporarily with the <code>sorted()</code> Function	44
Printing a List in Reverse Order	45
Finding the Length of a List	45
<i>Exercise 3-8: Seeing the World</i>	46
<i>Exercise 3-9: Dinner Guests</i>	46
<i>Exercise 3-10: Every Function</i>	46

Avoiding Index Errors When Working with Lists	46
Exercise 3-11: Intentional Error	48
Summary	48

4

WORKING WITH LISTS

49

Looping Through an Entire List	49
A Closer Look at Looping	50
Doing More Work Within a for Loop	51
Doing Something After a for Loop	52
Avoiding Indentation Errors	53
Forgetting to Indent	53
Forgetting to Indent Additional Lines	54
Indenting Unnecessarily	55
Indenting Unnecessarily After the Loop	55
Forgetting the Colon	56
Exercise 4-1: Pizzas	56
Exercise 4-2: Animals	56
Making Numerical Lists	57
Using the range() Function	57
Using range() to Make a List of Numbers	58
Simple Statistics with a List of Numbers	59
List Comprehensions	59
Exercise 4-3: Counting to Twenty	60
Exercise 4-4: One Million	60
Exercise 4-5: Summing a Million	60
Exercise 4-6: Odd Numbers	60
Exercise 4-7: Threes	60
Exercise 4-8: Cubes	60
Exercise 4-9: Cube Comprehension	60
Working with Part of a List	61
Slicing a List	61
Looping Through a Slice	62
Copying a List	63
Exercise 4-10: Slices	65
Exercise 4-11: My Pizzas, Your Pizzas	65
Exercise 4-12: More Loops	65
Tuples	65
Defining a Tuple	66
Looping Through All Values in a Tuple	67
Writing over a Tuple	67
Exercise 4-13: Buffet	68
Styling Your Code	68
The Style Guide	68
Indentation	69
Line Length	69
Blank Lines	69
Other Style Guidelines	70
Exercise 4-14: PEP 8	70
Exercise 4-15: Code Review	70
Summary	70

A Simple Example	72
Conditional Tests	72
Checking for Equality	72
Ignoring Case When Checking for Equality	73
Checking for Inequality	74
Numerical Comparisons	74
Checking Multiple Conditions	75
Checking Whether a Value Is in a List	76
Checking Whether a Value Is Not in a List	77
Boolean Expressions	77
<i>Exercise 5-1: Conditional Tests</i>	78
<i>Exercise 5-2: More Conditional Tests</i>	78
if Statements	78
Simple if Statements	78
if-else Statements	79
The if-elif-else Chain	80
Using Multiple elif Blocks	82
Omitting the else Block	82
Testing Multiple Conditions	83
<i>Exercise 5-3: Alien Colors #1</i>	84
<i>Exercise 5-4: Alien Colors #2</i>	84
<i>Exercise 5-5: Alien Colors #3</i>	85
<i>Exercise 5-6: Stages of Life</i>	85
<i>Exercise 5-7: Favorite Fruit</i>	85
Using if Statements with Lists	85
Checking for Special Items	86
Checking That a List Is Not Empty	87
Using Multiple Lists	88
<i>Exercise 5-8: Hello Admin</i>	89
<i>Exercise 5-9: No Users</i>	89
<i>Exercise 5-10: Checking Usernames</i>	89
<i>Exercise 5-11: Ordinal Numbers</i>	89
Styling Your if Statements	90
<i>Exercise 5-12: Styling if statements</i>	90
<i>Exercise 5-13: Your Ideas</i>	90
Summary	90

A Simple Dictionary	92
Working with Dictionaries	92
Accessing Values in a Dictionary	93
Adding New Key-Value Pairs	93
Starting with an Empty Dictionary	94
Modifying Values in a Dictionary	95
Removing Key-Value Pairs	96
A Dictionary of Similar Objects	97
Using get() to Access Values	98

<i>Exercise 6-1: Person</i>	99
<i>Exercise 6-2: Favorite Numbers</i>	99
<i>Exercise 6-3: Glossary</i>	99
Looping Through a Dictionary	99
Looping Through All Key-Value Pairs	99
Looping Through All the Keys in a Dictionary	101
Looping Through a Dictionary's Keys in a Particular Order	103
Looping Through All Values in a Dictionary	104
<i>Exercise 6-4: Glossary 2</i>	105
<i>Exercise 6-5: Rivers</i>	105
<i>Exercise 6-6: Polling</i>	105
Nesting	106
A List of Dictionaries	106
A List in a Dictionary	108
A Dictionary in a Dictionary	110
<i>Exercise 6-7: People</i>	112
<i>Exercise 6-8: Pets</i>	112
<i>Exercise 6-9: Favorite Places</i>	112
<i>Exercise 6-10: Favorite Numbers</i>	112
<i>Exercise 6-11: Cities</i>	112
<i>Exercise 6-12: Extensions</i>	112
Summary	112

7 USER INPUT AND WHILE LOOPS 113

How the <code>input()</code> Function Works	114
Writing Clear Prompts	114
Using <code>int()</code> to Accept Numerical Input	115
The Modulo Operator	116
<i>Exercise 7-1: Rental Car</i>	117
<i>Exercise 7-2: Restaurant Seating</i>	117
<i>Exercise 7-3: Multiples of Ten</i>	117
Introducing while Loops	118
The while Loop in Action	118
Letting the User Choose When to Quit	118
Using a Flag	120
Using <code>break</code> to Exit a Loop	121
Using <code>continue</code> in a Loop	122
Avoiding Infinite Loops	122
<i>Exercise 7-4: Pizza Toppings</i>	123
<i>Exercise 7-5: Movie Tickets</i>	123
<i>Exercise 7-6: Three Exits</i>	124
<i>Exercise 7-7: Infinity</i>	124
Using a while Loop with Lists and Dictionaries	124
Moving Items from One List to Another	124
Removing All Instances of Specific Values from a List	125
Filling a Dictionary with User Input	126
<i>Exercise 7-8: Deli</i>	127
<i>Exercise 7-9: No Pastrami</i>	127
<i>Exercise 7-10: Dream Vacation</i>	127
Summary	127

Defining a Function	130
Passing Information to a Function	130
Arguments and Parameters	131
<i>Exercise 8-1: Message</i>	131
<i>Exercise 8-2: Favorite Book</i>	131
Passing Arguments	131
Positional Arguments	132
Keyword Arguments	133
Default Values	134
Equivalent Function Calls	135
Avoiding Argument Errors	136
<i>Exercise 8-3: T-Shirt</i>	137
<i>Exercise 8-4: Large Shirts</i>	137
<i>Exercise 8-5: Cities</i>	137
Return Values	137
Returning a Simple Value	138
Making an Argument Optional	138
Returning a Dictionary	140
Using a Function with a while Loop	141
<i>Exercise 8-6: City Names</i>	142
<i>Exercise 8-7: Album</i>	142
<i>Exercise 8-8: User Albums</i>	142
Passing a List	143
Modifying a List in a Function	143
Preventing a Function from Modifying a List	145
<i>Exercise 8-9: Messages</i>	146
<i>Exercise 8-10: Sending Messages</i>	146
<i>Exercise 8-11: Archived Messages</i>	146
Passing an Arbitrary Number of Arguments	147
Mixing Positional and Arbitrary Arguments	148
Using Arbitrary Keyword Arguments	148
<i>Exercise 8-12: Sandwiches</i>	150
<i>Exercise 8-13: User Profile</i>	150
<i>Exercise 8-14: Cars</i>	150
Storing Your Functions in Modules	150
Importing an Entire Module	150
Importing Specific Functions	152
Using as to Give a Function an Alias	152
Using as to Give a Module an Alias	153
Importing All Functions in a Module	153
Styling Functions	154
<i>Exercise 8-15: Printing Models</i>	155
<i>Exercise 8-16: Imports</i>	155
<i>Exercise 8-17: Styling Functions</i>	155
Summary	155

Creating and Using a Class	158
Creating the Dog Class	158
Making an Instance from a Class	160
<i>Exercise 9-1: Restaurant</i>	162
<i>Exercise 9-2: Three Restaurants</i>	162
<i>Exercise 9-3: Users</i>	162
Working with Classes and Instances	162
The Car Class	162
Setting a Default Value for an Attribute	163
Modifying Attribute Values	164
<i>Exercise 9-4: Number Served</i>	167
<i>Exercise 9-5: Login Attempts</i>	167
Inheritance	167
The <code>__init__()</code> Method for a Child Class	167
Defining Attributes and Methods for the Child Class	169
Overriding Methods from the Parent Class	170
Instances as Attributes	170
Modeling Real-World Objects	173
<i>Exercise 9-6: Ice Cream Stand</i>	173
<i>Exercise 9-7: Admin</i>	173
<i>Exercise 9-8: Privileges</i>	173
<i>Exercise 9-9: Battery Upgrade</i>	174
Importing Classes	174
Importing a Single Class	174
Storing Multiple Classes in a Module	175
Importing Multiple Classes from a Module	177
Importing an Entire Module	177
Importing All Classes from a Module	177
Importing a Module into a Module	178
Using Aliases	179
Finding Your Own Workflow	179
<i>Exercise 9-10: Imported Restaurant</i>	180
<i>Exercise 9-11: Imported Admin</i>	180
<i>Exercise 9-12: Multiple Modules</i>	180
The Python Standard Library	180
<i>Exercise 9-13: Dice</i>	181
<i>Exercise 9-14: Lottery</i>	181
<i>Exercise 9-15: Lottery Analysis</i>	181
<i>Exercise 9-16: Python Module of the Week</i>	181
Styling Classes	181
Summary	182

Reading from a File	184
Reading an Entire File	184
File Paths	185
Reading Line by Line	187
Making a List of Lines from a File	188
Working with a File's Contents	188

Large Files: One Million Digits	189
Is Your Birthday Contained in Pi?	190
<i>Exercise 10-1: Learning Python</i>	191
<i>Exercise 10-2: Learning C</i>	191
Writing to a File.	191
Writing to an Empty File	191
Writing Multiple Lines	192
Appending to a File.	193
<i>Exercise 10-3: Guest</i>	193
<i>Exercise 10-4: Guest Book</i>	193
<i>Exercise 10-5: Programming Poll</i>	193
Exceptions.	194
Handling the ZeroDivisionError Exception	194
Using try-except Blocks	194
Using Exceptions to Prevent Crashes	195
The else Block.	196
Handling the FileNotFoundError Exception	197
Analyzing Text	198
Working with Multiple Files	199
Failing Silently	200
Deciding Which Errors to Report.	201
<i>Exercise 10-6: Addition</i>	201
<i>Exercise 10-7: Addition Calculator</i>	202
<i>Exercise 10-8: Cats and Dogs</i>	202
<i>Exercise 10-9: Silent Cats and Dogs</i>	202
<i>Exercise 10-10: Common Words</i>	202
Storing Data	202
Using <code>json.dump()</code> and <code>json.load()</code>	203
Saving and Reading User-Generated Data	204
Refactoring.	206
<i>Exercise 10-11: Favorite Number</i>	208
<i>Exercise 10-12: Favorite Number Remembered</i>	208
<i>Exercise 10-13: Verify User</i>	208
Summary	208

11		209
TESTING YOUR CODE		
Testing a Function	210	
Unit Tests and Test Cases	211	
A Passing Test	211	
A Failing Test	212	
Responding to a Failed Test	213	
Adding New Tests.	214	
<i>Exercise 11-1: City, Country</i>	215	
<i>Exercise 11-2: Population</i>	216	
Testing a Class	216	
A Variety of Assert Methods	216	
A Class to Test	217	
Testing the AnonymousSurvey Class	218	
The <code>setUp()</code> Method	220	
<i>Exercise 11-3: Employee</i>	221	
Summary	222	

PROJECT 1: ALIEN INVASION

12		
A SHIP THAT FIRES BULLETS		227
Planning Your Project	228	
Installing Pygame	228	
Starting the Game Project	229	
Creating a Pygame Window and Responding to User Input	229	
Setting the Background Color	230	
Creating a Settings Class	231	
Adding the Ship Image	232	
Creating the Ship Class	233	
Drawing the Ship to the Screen	235	
Refactoring: The <code>_check_events()</code> and <code>_update_screen()</code> Methods	236	
The <code>_check_events()</code> Method	236	
The <code>_update_screen()</code> Method	237	
<i>Exercise 12-1: Blue Sky</i>	238	
<i>Exercise 12-2: Game Character</i>	238	
Piloting the Ship	238	
Responding to a Keypress	238	
Allowing Continuous Movement	239	
Moving Both Left and Right	240	
Adjusting the Ship's Speed	241	
Limiting the Ship's Range	243	
Refactoring <code>_check_events()</code>	243	
Pressing Q to Quit	244	
Running the Game in Fullscreen Mode	244	
A Quick Recap	245	
<code>alien_invasion.py</code>	245	
<code>settings.py</code>	246	
<code>ship.py</code>	246	
<i>Exercise 12-3: Pygame Documentation</i>	246	
<i>Exercise 12-4: Rocket</i>	246	
<i>Exercise 12-5: Keys</i>	246	
Shooting Bullets	246	
Adding the Bullet Settings	247	
Creating the Bullet Class	247	
Storing Bullets in a Group	248	
Firing Bullets	249	
Deleting Old Bullets	250	
Limiting the Number of Bullets	251	
Creating the <code>_update_bullets()</code> Method	252	
<i>Exercise 12-6: Sideways Shooter</i>	253	
Summary	253	

13	ALIENS!	255
Reviewing the Project	256	
Creating the First Alien	256	
Creating the Alien Class	257	
Creating an Instance of the Alien	258	
Building the Alien Fleet	259	
Determining How Many Aliens Fit in a Row	260	
Creating a Row of Aliens	260	
Refactoring <code>_create_fleet()</code>	262	
Adding Rows	262	
<i>Exercise 13-1: Stars</i>	264	
<i>Exercise 13-2: Better Stars</i>	264	
Making the Fleet Move	265	
Moving the Aliens Right	265	
Creating Settings for Fleet Direction	266	
Checking Whether an Alien Has Hit the Edge	266	
Dropping the Fleet and Changing Direction	267	
<i>Exercise 13-3: Raindrops</i>	268	
<i>Exercise 13-4: Steady Rain</i>	268	
Shooting Aliens	268	
Detecting Bullet Collisions	268	
Making Larger Bullets for Testing	270	
Repopulating the Fleet	270	
Speeding Up the Bullets	271	
Refactoring <code>_update_bullets()</code>	271	
<i>Exercise 13-5: Sideways Shooter Part 2</i>	272	
Ending the Game	272	
Detecting Alien and Ship Collisions	272	
Responding to Alien and Ship Collisions	273	
Aliens that Reach the Bottom of the Screen	276	
Game Over!	276	
Identifying When Parts of the Game Should Run	277	
<i>Exercise 13-6: Game Over</i>	278	
Summary	278	
14	SCORING	279
Adding the Play Button	280	
Creating a Button Class	280	
Drawing the Button to the Screen	281	
Starting the Game	283	
Resetting the Game	283	
Deactivating the Play Button	284	
Hiding the Mouse Cursor	284	
<i>Exercise 14-1: Press P to Play</i>	285	
<i>Exercise 14-2: Target Practice</i>	285	
Leveling Up	285	
Modifying the Speed Settings	285	
Resetting the Speed	287	

Exercise 14-3: Challenging Target Practice	288
Exercise 14-4: Difficulty Levels	288
Scoring	288
Displaying the Score	288
Making a Scoreboard	289
Updating the Score as Aliens Are Shot Down	291
Resetting the Score	291
Making Sure to Score All Hits	292
Increasing Point Values	292
Rounding the Score	293
High Scores	294
Displaying the Level	296
Displaying the Number of Ships	298
Exercise 14-5: All-Time High Score	301
Exercise 14-6: Refactoring	301
Exercise 14-7: Expanding the Game	302
Exercise 14-8: Sideways Shooter, Final Version	302
Summary	302

PROJECT 2: DATA VISUALIZATION

15		305
GENERATING DATA		
Installing Matplotlib	306	
Plotting a Simple Line Graph	306	
Changing the Label Type and Line Thickness	307	
Correcting the Plot	309	
Using Built-in Styles	310	
Plotting and Styling Individual Points with scatter()	310	
Plotting a Series of Points with scatter()	312	
Calculating Data Automatically	312	
Defining Custom Colors	314	
Using a Colormap	314	
Saving Your Plots Automatically	315	
Exercise 15-1: Cubes	315	
Exercise 15-2: Colored Cubes	315	
Random Walks	315	
Creating the RandomWalk() Class	316	
Choosing Directions	316	
Plotting the Random Walk	317	
Generating Multiple Random Walks	318	
Styling the Walk	319	
Exercise 15-3: Molecular Motion	323	
Exercise 15-4: Modified Random Walks	323	
Exercise 15-5: Refactoring	323	
Rolling Dice with Plotly	323	
Installing Plotly	324	
Creating the Die Class	324	
Rolling the Die	325	

Analyzing the Results	325
Making a Histogram	326
Rolling Two Dice	328
Rolling Dice of Different Sizes	329
Exercise 15-6: Two D8s	331
Exercise 15-7: Three Dice	331
Exercise 15-8: Multiplication	331
Exercise 15-9: Die Comprehensions	331
Exercise 15-10: Practicing with Both Libraries	331
Summary	331

16 DOWNLOADING DATA 333

The CSV File Format	334
Parsing the CSV File Headers	334
Printing the Headers and Their Positions	335
Extracting and Reading Data	336
Plotting Data in a Temperature Chart	336
The <code>datetime</code> Module	337
Plotting Dates	338
Plotting a Longer Timeframe	340
Plotting a Second Data Series	340
Shading an Area in the Chart	342
Error Checking	343
Downloading Your Own Data	345
Exercise 16-1: Sitka Rainfall	346
Exercise 16-2: Sitka–Death Valley Comparison	346
Exercise 16-3: San Francisco	346
Exercise 16-4: Automatic Indexes	347
Exercise 16-5: Explore	347
Mapping Global Data Sets: JSON Format	347
Downloading Earthquake Data	347
Examining JSON Data	347
Making a List of All Earthquakes	350
Extracting Magnitudes	350
Extracting Location Data	351
Building a World Map	351
A Different Way of Specifying Chart Data	353
Customizing Marker Size	353
Customizing Marker Colors	354
Other Colorscales	356
Adding Hover Text	356
Exercise 16-6: Refactoring	357
Exercise 16-7: Automated Title	357
Exercise 16-8: Recent Earthquakes	358
Exercise 16-9: World Fires	358
Summary	358

17	WORKING WITH APIs	359
Using a Web API	359	
Git and GitHub	360	
Requesting Data Using an API Call	360	
Installing Requests	361	
Processing an API Response	361	
Working with the Response Dictionary	362	
Summarizing the Top Repositories	364	
Monitoring API Rate Limits	365	
Visualizing Repositories Using Plotly	366	
Refining Plotly Charts	368	
Adding Custom Tooltips	369	
Adding Clickable Links to Our Graph	370	
More About Plotly and the GitHub API	371	
The Hacker News API	372	
Exercise 17-1: Other Languages	375	
Exercise 17-2: Active Discussions	375	
Exercise 17-3: Testing <code>python_repos.py</code>	375	
Exercise 17-4: Further Exploration	375	
Summary	375	

PROJECT 3: WEB APPLICATIONS

18	GETTING STARTED WITH DJANGO	379
Setting Up a Project	380	
Writing a Spec	380	
Creating a Virtual Environment	380	
Activating the Virtual Environment	381	
Installing Django	381	
Creating a Project in Django	382	
Creating the Database	382	
Viewing the Project	383	
Exercise 18-1: New Projects	384	
Starting an App	384	
Defining Models	385	
Activating Models	386	
The Django Admin Site	387	
Defining the Entry Model	390	
Migrating the Entry Model	391	
Registering Entry with the Admin Site	391	
The Django Shell	392	
Exercise 18-2: Short Entries	394	
Exercise 18-3: The Django API	394	
Exercise 18-4: Pizzeria	394	
Making Pages: The Learning Log Home Page	394	
Mapping a URL	395	
Writing a View	396	

Writing a Template	397
Exercise 18-5: Meal Planner.	398
Exercise 18-6: Pizzeria Home Page.	398
Building Additional Pages	398
Template Inheritance	398
The Topics Page	400
Individual Topic Pages.	403
Exercise 18-7: Template Documentation	406
Exercise 18-8: Pizzeria Pages.	406
Summary	407

19 **USER ACCOUNTS** **409**

Allowing Users to Enter Data	410
Adding New Topics	410
Adding New Entries	414
Editing Entries.	418
Exercise 19-1: Blog	421
Setting Up User Accounts	421
The users App.	421
The Login Page	422
Logging Out	424
The Registration Page	426
Exercise 19-2: Blog Accounts	428
Allowing Users to Own Their Data	428
Restricting Access with @login_required.	429
Connecting Data to Certain Users	430
Restricting Topics Access to Appropriate Users	433
Protecting a User's Topics	434
Protecting the edit_entry Page.	434
Associating New Topics with the Current User	435
Exercise 19-3: Refactoring	436
Exercise 19-4: Protecting new_entry	436
Exercise 19-5: Protected Blog	436
Summary	436

20 **STYLING AND DEPLOYING AN APP** **437**

Styling Learning Log	438
The django-bootstrap4 App	438
Using Bootstrap to Style Learning Log	438
Modifying base.html	439
Styling the Home Page Using a Jumbotron	443
Styling the Login Page	444
Styling the Topics Page	445
Styling the Entries on the Topic Page	446
Exercise 20-1: Other Forms	447
Exercise 20-2: Stylish Blog	447
Deploying Learning Log	448
Making a Heroku Account	448
Installing the Heroku CLI.	448

Installing Required Packages	448
Creating a requirements.txt File	448
Specifying the Python Runtime	449
Modifying settings.py for Heroku	450
Making a Procfile to Start Processes	450
Using Git to Track the Project's Files	450
Pushing to Heroku	452
Setting Up the Database on Heroku	454
Refining the Heroku Deployment	454
Securing the Live Project	456
Committing and Pushing Changes	457
Setting Environment Variables on Heroku	458
Creating Custom Error Pages	458
Ongoing Development	461
The SECRET_KEY Setting	461
Deleting a Project on Heroku	461
<i>Exercise 20-3: Live Blog</i>	462
<i>Exercise 20-4: More 404s</i>	462
<i>Exercise 20-5: Extended Learning Log</i>	462
Summary	463

AFTERWORD **465**

A	
INSTALLATION AND TROUBLESHOOTING	467
Python on Windows	467
Finding the Python Interpreter	467
Adding Python to Your Path Variable	468
Reinstalling Python	469
Python on macOS	469
Installing Homebrew	469
Installing Python	470
Python on Linux	470
Python Keywords and Built-in Functions	471
Python Keywords	471
Python Built-in Functions	471

B **TEXT EDITORS AND IDES** **473**

Customizing Sublime Text Settings	474
Converting Tabs to Spaces	474
Setting the Line Length Indicator	474
Indenting and Unindenting Code Blocks	474
Commenting Out Blocks of Code	475
Saving Your Configuration	475
Further Customizations	475
Other Text Editors and IDEs	475
IDLE	475
Geany	476
Emacs and Vim	476

C		
GETTING HELP		479
First Steps	479	
Try It Again	480	
Take a Break	480	
Refer to This Book's Resources	480	
Searching Online	481	
Stack Overflow	481	
The Official Python Documentation	481	
Official Library Documentation	482	
r/learnpython	482	
Blog Posts	482	
Internet Relay Chat	482	
Making an IRC Account	482	
Channels to Join	483	
IRC Culture	483	
Slack	483	
Discord	484	
D		
USING GIT FOR VERSION CONTROL		485
Installing Git	486	
Installing Git on Windows	486	
Installing Git on macOS	486	
Installing Git on Linux	486	
Configuring Git	486	
Making a Project	486	
Ignoring Files	487	
Initializing a Repository	487	
Checking the Status	487	
Adding Files to the Repository	488	
Making a Commit	488	
Checking the Log	489	
The Second Commit	489	
Reverting a Change	490	
Checking Out Previous Commits	491	
Deleting the Repository	493	
INDEX		495

PREFACE TO THE SECOND EDITION

The response to the first edition of *Python Crash Course* has been overwhelmingly positive. More than 500,000 copies are in print, including translations in eight languages. I've received letters and emails from readers as young as 10, as well as from retirees who want to learn to program in their free time. *Python Crash Course* is being used in middle schools and high schools, and also in college classes. Students who are assigned more advanced textbooks are using *Python Crash Course* as a companion text for their classes and finding it a worthwhile supplement. People are using it to enhance their skills on the job and to start working on their own side projects. In short, people are using the book for the full range of purposes I had hoped they would.

The opportunity to write a second edition of *Python Crash Course* has been thoroughly enjoyable. Although Python is a mature language, it continues to evolve as every language does. My goal in revising the book was to make it leaner and simpler. There is no longer any reason to learn Python 2, so this edition focuses on Python 3 only. Many Python packages have become easier to install, so setup and installation instructions are easier. I've added a few topics that I've realized readers would benefit from, and I've updated some sections to reflect new, simpler ways of doing things in Python. I've also clarified some sections where certain details of the

language were not presented as accurately as they could have been. All the projects have been completely updated using popular, well-maintained libraries that you can confidently use to build your own projects.

The following is a summary of specific changes that have been made in the second edition:

- In Chapter 1, the instructions for installing Python have been simplified for users of all major operating systems. I now recommend the text editor Sublime Text, which is popular among beginner and professional programmers and works well on all operating systems.
- Chapter 2 includes a more accurate description of how variables are implemented in Python. Variables are described as *labels* for values, which leads to a better understanding of how variables behave in Python. The book now uses f-strings, introduced in Python 3.6. This is a much simpler way to use variable values in strings. The use of underscores to represent large numbers, such as `1_000_000`, was also introduced in Python 3.6 and is included in this edition. Multiple assignment of variables was previously introduced in one of the projects, and that description has been generalized and moved to Chapter 2 for the benefit of all readers. Finally, a clear convention for representing constant values in Python is included in this chapter.
- In Chapter 6, I introduce the `get()` method for retrieving values from a dictionary, which can return a default value if a key does not exist.
- The Alien Invasion project (Chapters 12–14) is now entirely class-based. The game itself is a class, rather than a series of functions. This greatly simplifies the overall structure of the game, vastly reducing the number of function calls and parameters required. Readers familiar with the first edition will appreciate the simplicity this new class-based approach provides. Pygame can now be installed in one line on all systems, and readers are given the option of running the game in fullscreen mode or in a windowed mode.
- In the data visualization projects, the installation instructions for Matplotlib are simpler for all operating systems. The visualizations featuring Matplotlib use the `subplots()` function, which will be easier to build upon as you learn to create more complex visualizations. The Rolling Dice project in Chapter 15 uses Plotly, a well-maintained visualization library that features a clean syntax and beautiful, fully customizable output.
- In Chapter 16, the weather project is based on data from NOAA, which should be more stable over the next few years than the site used in the first edition. The mapping project focuses on global earthquake activity; by the end of this project you'll have a stunning visualization showing Earth's tectonic plate boundaries through a focus on the locations of all earthquakes over a given time period. You'll learn to plot any data set involving geographic points.
- Chapter 17 uses Plotly to visualize Python-related activity in open source projects on GitHub.

- The Learning Log project (Chapters 18–20) is built using the latest version of Django and styled using the latest version of Bootstrap. The process of deploying the project to Heroku has been simplified using the `django-heroku` package, and uses environment variables rather than modifying the `settings.py` files. This is a simpler approach and is more consistent with how professional programmers deploy modern Django projects.
- Appendix A has been fully updated to recommend current best practices in installing Python. Appendix B includes detailed instructions for setting up Sublime Text and brief descriptions of most of the major text editors and IDEs in current use. Appendix C directs readers to newer, more popular online resources for getting help, and Appendix D continues to offer a mini crash course in using Git for version control.
- The index has been thoroughly updated to allow you to use *Python Crash Course* as a reference for all of your future Python projects.

Thank you for reading *Python Crash Course*! If you have any feedback or questions, please feel free to get in touch.

ACKNOWLEDGMENTS

This book would not have been possible without the wonderful and extremely professional staff at No Starch Press. Bill Pollock invited me to write an introductory book, and I deeply appreciate that original offer. Tyler Ortman helped shape my thinking in the early stages of drafting. Liz Chadwick's and Leslie Shen's initial feedback on each chapter was invaluable, and Anne Marie Walker helped to clarify many parts of the book. Riley Hoffman answered every question I had about the process of assembling a complete book and patiently turned my work into a beautiful finished product.

I'd like to thank Kenneth Love, the technical reviewer for *Python Crash Course*. I met Kenneth at PyCon one year, and his enthusiasm for the language and the Python community has been a constant source of professional inspiration ever since. Kenneth went beyond simple fact-checking and reviewed the book with the goal of helping beginning programmers develop a solid understanding of the Python language and programming in general. That said, any inaccuracies that remain are completely my own.

I'd like to thank my father for introducing me to programming at a young age and for not being afraid that I'd break his equipment. I'd like to thank my wife, Erin, for supporting and encouraging me through the writing of this book, and I'd like to thank my son, Ever, whose curiosity inspires me every single day.

INTRODUCTION



Every programmer has a story about how they learned to write their first program. I started programming as a child when my father was working for Digital Equipment Corporation, one of the pioneering companies of the modern computing era. I wrote my first program on a kit computer that my dad had assembled in our basement. The computer consisted of nothing more than a bare motherboard connected to a keyboard without a case, and its monitor was a bare cathode ray tube. My initial program was a simple number guessing game, which looked something like this:

```
I'm thinking of a number! Try to guess the number I'm thinking of: 25
Too low! Guess again: 50
Too high! Guess again: 42
That's it! Would you like to play again? (yes/no) no
Thanks for playing!
```

I'll always remember how satisfied I felt watching my family play a game that I created and that worked as I intended it to.

That early experience had a lasting impact. There is real satisfaction in building something with a purpose, something that solves a problem. The software I write now meets a more significant need than my childhood efforts, but the sense of satisfaction I get from creating a program that works is still largely the same.

Who Is This Book For?

The goal of this book is to bring you up to speed with Python as quickly as possible so you can build programs that work—games, data visualizations, and web applications—while developing a foundation in programming that will serve you well for the rest of your life. *Python Crash Course* is written for people of any age who have never before programmed in Python or have never programmed at all. This book is for those who want to learn the basics of programming quickly so they can focus on interesting projects, and those who like to test their understanding of new concepts by solving meaningful problems. *Python Crash Course* is also perfect for middle school and high school teachers who want to offer their students a project-based introduction to programming. If you're taking a college class and want a friendlier introduction to Python than the text you've been assigned, this book could make your class easier as well.

What Can You Expect to Learn?

The purpose of this book is to make you a good programmer in general and a good Python programmer in particular. You'll learn efficiently and adopt good habits as I provide you with a solid foundation in general programming concepts. After working your way through *Python Crash Course*, you should be ready to move on to more advanced Python techniques, and your next programming language will be even easier to grasp.

In the first part of this book, you'll learn basic programming concepts you need to know to write Python programs. These concepts are the same as those you'd learn when starting out in almost any programming language. You'll learn about different kinds of data and the ways you can store data in lists and dictionaries within your programs. You'll learn to build collections of data and work through those collections in efficient ways. You'll learn to use `while` loops and `if` statements to test for certain conditions so you can run specific sections of code while those conditions are true and run other sections when they're not—a technique that greatly helps you automate processes.

You'll learn to accept input from users to make your programs interactive and to keep your programs running as long as the user is active. You'll explore how to write functions to make parts of your program reusable, so you only have to write blocks of code that perform certain actions once and then use that code as many times as you like. You'll then extend

this concept to more complicated behavior with classes, making fairly simple programs respond to a variety of situations. You’ll learn to write programs that handle common errors gracefully. After working through each of these basic concepts, you’ll write a few short programs that solve some well-defined problems. Finally, you’ll take your first step toward intermediate programming by learning how to write tests for your code so you can develop your programs further without worrying about introducing bugs. All the information in Part I will prepare you for taking on larger, more complex projects.

In Part II, you’ll apply what you learned in Part I to three projects. You can do any or all of these projects in whichever order works best for you. In the first project (Chapters 12–14), you’ll create a *Space Invaders*–style shooting game called *Alien Invasion*, which consists of levels of increasing difficulty. After you’ve completed this project, you should be well on your way to being able to develop your own 2D games.

The second project (Chapters 15–17) introduces you to data visualization. Data scientists aim to make sense of the vast amount of information available to them through a variety of visualization techniques. You’ll work with data sets that you generate through code, data sets that you download from online sources, and data sets your programs download automatically. After you’ve completed this project, you’ll be able to write programs that sift through large data sets and make visual representations of that stored information.

In the third project (Chapters 18–20), you’ll build a small web application called Learning Log. This project allows you to keep a journal of ideas and concepts you’ve learned about a specific topic. You’ll be able to keep separate logs for different topics and allow others to create an account and start their own journals. You’ll also learn how to deploy your project so anyone can access it online from anywhere.

Online Resources

You can find all the supplementary resources for the book online at <https://nostarch.com/pythoncrashcourse2e/> or http://ehmatthes.github.io/pcc_2e/. These resources include:

Setup instructions These instructions are identical to what’s in the book but include active links you can click for all the different pieces. If you’re having any setup issues, refer to this resource.

Updates Python, like all languages, is constantly evolving. I maintain a thorough set of updates, so if anything isn’t working, check here to see whether instructions have changed.

Solutions to exercises You should spend significant time on your own attempting the exercises in the “Try It Yourself” sections. But if you’re stuck and can’t make any progress, solutions to most of the exercises are online.

Cheat sheets A full set of downloadable cheat sheets for a quick reference to major concepts is also online.

Why Python?

Every year I consider whether to continue using Python or whether to move on to a different language—perhaps one that’s newer to the programming world. But I continue to focus on Python for many reasons. Python is an incredibly efficient language: your programs will do more in fewer lines of code than many other languages would require. Python’s syntax will also help you write “clean” code. Your code will be easy to read, easy to debug, and easy to extend and build upon compared to other languages.

People use Python for many purposes: to make games, build web applications, solve business problems, and develop internal tools at all kinds of interesting companies. Python is also used heavily in scientific fields for academic research and applied work.

One of the most important reasons I continue to use Python is because of the Python community, which includes an incredibly diverse and welcoming group of people. Community is essential to programmers because programming isn’t a solitary pursuit. Most of us, even the most experienced programmers, need to ask advice from others who have already solved similar problems. Having a well-connected and supportive community is critical in helping you solve problems, and the Python community is fully supportive of people like you who are learning Python as your first programming language.

Python is a great language to learn, so let’s get started!

PART I

BASICS

Part I of this book teaches you the basic concepts you'll need to write Python programs. Many of these concepts are common to all programming languages, so they'll be useful throughout your life as a programmer.

In **Chapter 1** you'll install Python on your computer and run your first program, which prints the message *Hello world!* to the screen.

In **Chapter 2** you'll learn to store information in variables and work with text and numerical values.

Chapters 3 and **4** introduce lists. Lists can store as much information as you want in one variable, allowing you to work with that data efficiently. You'll be able to work with hundreds, thousands, and even millions of values in just a few lines of code.

In **Chapter 5** you'll use `if` statements to write code that responds one way if certain conditions are true, and responds in a different way if those conditions are not true.

Chapter 6 shows you how to use Python's dictionaries, which let you make connections between different pieces of information. Like lists, dictionaries can contain as much information as you need to store.

In **Chapter 7** you'll learn how to accept input from users to make your programs interactive. You'll also learn about `while` loops, which run blocks of code repeatedly as long as certain conditions remain true.

In **Chapter 8** you'll write functions, which are named blocks of code that perform a specific task and can be run whenever you need them.

Chapter 9 introduces classes, which allow you to model real-world objects, such as dogs, cats, people, cars, rockets, and much more, so your code can represent anything real or abstract.

Chapter 10 shows you how to work with files and handle errors so your programs won't crash unexpectedly. You'll store data before your program closes, and read the data back in when the program runs again. You'll learn about Python's exceptions, which allow you to anticipate errors, and make your programs handle those errors gracefully.

In **Chapter 11** you'll learn to write tests for your code to check that your programs work the way you intend them to. As a result, you'll be able to expand your programs without worrying about introducing new bugs. Testing your code is one of the first skills that will help you transition from beginner to intermediate programmer.

1

GETTING STARTED



In this chapter, you'll run your first Python program, *hello_world.py*. First, you'll need to check whether a recent version of Python is installed on your computer; if it isn't, you'll install it. You'll also install a text editor to work with your Python programs. Text editors recognize Python code and highlight sections as you write, making it easy to understand your code's structure.

Setting Up Your Programming Environment

Python differs slightly on different operating systems, so you'll need to keep a few considerations in mind. In the following sections, we'll make sure Python is set up correctly on your system.

Python Versions

Every programming language evolves as new ideas and technologies emerge, and the developers of Python have continually made the language more versatile and powerful. As of this writing, the latest version is Python 3.7, but everything in this book should run on Python 3.6 or later. In this section, we'll find out if Python is already installed on your system and whether you need to install a newer version. Appendix A contains a comprehensive guide to installing the latest version of Python on each major operating system as well.

Some old Python projects still use Python 2, but you should use Python 3. If Python 2 is installed on your system, it's probably there to support some older programs that your system needs. We'll leave this installation as is, and make sure you have a more recent version to work with.

Running Snippets of Python Code

You can run Python's interpreter in a terminal window, allowing you to try bits of Python code without having to save and run an entire program.

Throughout this book, you'll see code snippets that look like this:

```
❶ >>> print("Hello Python interpreter!")  
Hello Python interpreter!
```

The `>>>` prompt indicates that you should be using the terminal window, and the bold text is the code you should type in and then execute by pressing ENTER. Most of the examples in the book are small, self-contained programs that you'll run from your text editor rather than the terminal, because you'll write most of your code in the text editor. But sometimes basic concepts will be shown in a series of snippets run through a Python terminal session to demonstrate particular concepts more efficiently. When you see three angle brackets in a code listing ❶, you're looking at code and output from a terminal session. We'll try coding in the interpreter on your system in a moment.

We'll also use a text editor to create a simple program called *Hello World!* that has become a staple of learning to program. There's a long-held tradition in the programming world that printing a *Hello world!* message to the screen as your first program in a new language will bring you good luck. Such a simple program serves a very real purpose. If it runs correctly on your system, any Python program you write should work as well.

About the Sublime Text Editor

Sublime Text is a simple text editor that can be installed on all modern operating systems. Sublime Text lets you run almost all of your programs directly from the editor instead of through a terminal. Your code runs in a terminal session embedded in the Sublime Text window, which makes it easy to see the output.

Sublime Text is a beginner-friendly editor, but many professional programmers use it as well. If you become comfortable using it while learning Python, you can continue using it as you progress to larger and more complicated projects. Sublime Text has a very liberal licensing policy: you can use the editor free of charge as long as you want, but the developers request that you purchase a license if you like it and want to keep using it.

Appendix B provides information on other text editors. If you’re curious about the other options, you might want to skim that appendix at this point. If you want to begin programming quickly, you can use Sublime Text to start and consider other editors once you’ve gained some experience as a programmer. In this chapter, I’ll walk you through installing Sublime Text on your operating system.

Python on Different Operating Systems

Python is a cross-platform programming language, which means it runs on all the major operating systems. Any Python program you write should run on any modern computer that has Python installed. However, the methods for setting up Python on different operating systems vary slightly.

In this section, you’ll learn how to set up Python on your system. You’ll first check whether a recent version of Python is installed on your system and install it if it’s not. Then you’ll install Sublime Text. These are the only two steps that are different for each operating system.

In the sections that follow, you’ll run the *Hello World!* program and troubleshoot anything that didn’t work. I’ll walk you through this process for each operating system, so you’ll have a beginner-friendly Python programming environment.

Python on Windows

Windows doesn’t always come with Python, so you’ll probably need to install it, and then install Sublime Text.

Installing Python

First, check whether Python is installed on your system. Open a command window by entering `command` into the Start menu or by holding down the SHIFT key while right-clicking on your desktop and selecting **Open command window here** from the menu. In the terminal window, enter `python` in lowercase. If you get a Python prompt (`>>>`) in response, Python is installed on your system. If you see an error message telling you that `python` is not a recognized command, Python isn’t installed.

In that case, or if you see a version of Python earlier than Python 3.6, you need to download a Python installer for Windows. Go to <https://python.org/> and hover over the **Downloads** link. You should see a button for downloading the latest version of Python. Click the button, which should automatically start downloading the correct installer for your system. After

you've downloaded the file, run the installer. Make sure you select the option **Add Python to PATH**, which will make it easier to configure your system correctly. Figure 1-1 shows this option selected.

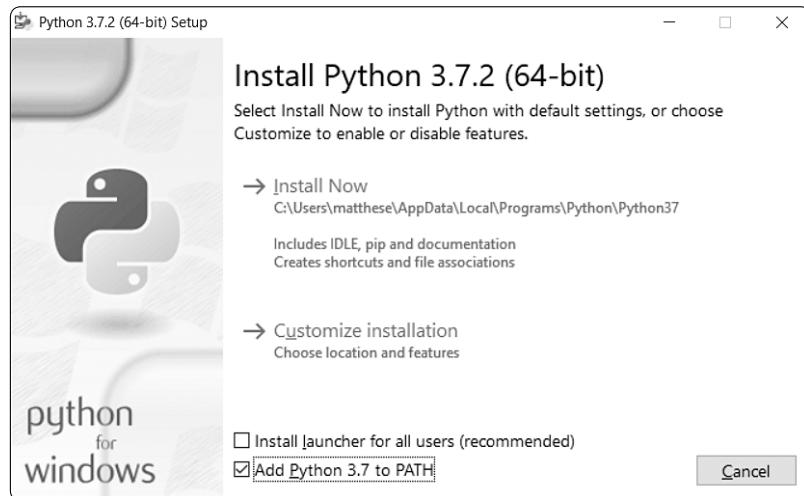


Figure 1-1: Make sure you select the checkbox labeled Add Python to PATH.

Running Python in a Terminal Session

Open a command window and enter `python` in lowercase. You should see a Python prompt (`>>>`), which means Windows has found the version of Python you just installed.

```
C:\> python
Python 3.7.2 (v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

NOTE If you don't see this output or something similar, see the more detailed setup instructions in Appendix A.

Enter the following line in your Python session, and make sure you see the output Hello Python interpreter!

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Any time you want to run a snippet of Python code, open a command window and start a Python terminal session. To close the terminal session, press CTRL-Z and then press ENTER, or enter the command `exit()`.

Installing Sublime Text

You can download an installer for Sublime Text at <https://sublimetext.com/>. Click the download link and look for a Windows installer. After downloading the installer, run the installer and accept all of its defaults.

Python on macOS

Python is already installed on most macOS systems, but it's most likely an outdated version that you won't want to learn on. In this section, you'll install the latest version of Python, and then you'll install Sublime Text and make sure it's configured correctly.

Checking Whether Python 3 Is Installed

Open a terminal window by going to **Applications ▶ Utilities ▶ Terminal**. You can also press ⌘-spacebar, type **terminal**, and then press ENTER. To see which version of Python is installed, enter **python** with a lowercase *p*—this also starts the Python interpreter within the terminal, allowing you to enter Python commands. You should see output telling you which Python version is installed on your system and a **>>>** prompt where you can start entering Python commands, like this:

```
$ python
Python 2.7.15 (default, Aug 17 2018, 22:39:05)
[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>>
```

This output indicates that Python 2.7.15 is currently the default version installed on this computer. Once you've seen this output, press CTRL-D or enter **exit()** to leave the Python prompt and return to a terminal prompt.

To check whether you have Python 3 installed, enter the command **python3**. You'll probably get an error message, meaning you don't have any versions of Python 3 installed. If the output shows you have Python 3.6 or a later version installed, you can skip ahead to “Running Python in a Terminal Session” on page 8. If Python 3 isn't installed by default, you'll need to install it manually. Note that whenever you see the **python** command in this book, you need to use the **python3** command instead to make sure you're using Python 3, not Python 2; they differ significantly enough that you'll run into trouble trying to run the code in this book using Python 2.

If you see any version earlier than Python 3.6, follow the instructions in the next section to install the latest version.

Installing the Latest Version of Python

You can find a Python installer for your system at <https://python.org/>. Hover over the **Download** link, and you should see a button for downloading the latest Python version. Click the button, which should automatically start

downloading the correct installer for your system. After the file downloads, run the installer.

When you're finished, enter the following at a terminal prompt:

```
$ python3 --version  
Python 3.7.2
```

You should see output similar to this, in which case, you're ready to try out Python. Whenever you see the command `python`, make sure you use `python3`.

Running Python in a Terminal Session

You can now try running snippets of Python code by opening a terminal and typing `python3`. Enter the following line in the terminal session:

```
>>> print("Hello Python interpreter!")  
Hello Python interpreter!  
>>>
```

Your message should print directly in the current terminal window. Remember that you can close the Python interpreter by pressing CTRL-D or by entering the command `exit()`.

Installing Sublime Text

To install the Sublime Text editor, you need to download the installer at <https://sublimetext.com/>. Click the **Download** link and look for an installer for macOS. After the installer downloads, open it and then drag the Sublime Text icon into your *Applications* folder.

Python on Linux

Linux systems are designed for programming, so Python is already installed on most Linux computers. The people who write and maintain Linux expect you to do your own programming at some point and encourage you to do so. For this reason, there's very little to install and only a few settings to change to start programming.

Checking Your Version of Python

Open a terminal window by running the Terminal application on your system (in Ubuntu, you can press CTRL-ALT-T). To find out which version of Python is installed, enter `python3` with a lowercase *p*. When Python is installed, this command starts the Python interpreter. You should see output indicating which version of Python is installed and a `>>>` prompt where you can start entering Python commands, like this:

```
$ python3  
Python 3.7.2 (default, Dec 27 2018, 04:01:51)  
[GCC 7.3.0] on linux
```

```
Type "help", "copyright", "credits" or "license" for more information.  
>>>
```

This output indicates that Python 3.7.2 is currently the default version of Python installed on this computer. When you've seen this output, press CTRL-D or enter `exit()` to leave the Python prompt and return to a terminal prompt. Whenever you see the `python` command in this book, enter `python3` instead.

You'll need Python 3.6 or later to run the code in this book. If the Python version installed on your system is earlier than Python 3.6, refer to Appendix A to install the latest version.

Running Python in a Terminal Session

You can try running snippets of Python code by opening a terminal and entering `python3`, as you did when checking your version. Do this again, and when you have Python running, enter the following line in the terminal session:

```
>>> print("Hello Python interpreter!")  
Hello Python interpreter!  
>>>
```

The message should print directly in the current terminal window. Remember that you can close the Python interpreter by pressing CTRL-D or by entering the command `exit()`.

Installing Sublime Text

On Linux, you can install Sublime Text from the Ubuntu Software Center. Click the Ubuntu Software icon in your menu, and search for **Sublime Text**. Click to install it, and then launch it.

Running a Hello World Program

With a recent version of Python and Sublime Text installed, you're almost ready to run your first Python program written in a text editor. But before doing so, you need to make sure Sublime Text is set up to use the correct version of Python on your system. Then you'll write the *Hello World!* program and run it.

Configuring Sublime Text to Use the Correct Python Version

If the `python` command on your system runs Python 3, you won't need to configure anything and can skip to the next section. If you use the `python3` command, you'll need to configure Sublime Text to use the correct Python version when it runs your programs.

Click the Sublime Text icon to launch it, or search for Sublime Text in your system’s search bar and then launch it. Go to **Tools** ▶ **Build System** ▶ **New Build System**, which will open a new configuration file for you. Delete what you see and enter the following:

```
Python3
.sublime-build
{
    "cmd": ["python3", "-u", "$file"],
}
```

This code tells Sublime Text to use your system’s `python3` command when running your Python program files. Save the file as `Python3.sublime-build` in the default directory that Sublime Text opens when you choose Save.

Running hello_world.py

Before you write your first program, make a folder called `python_work` somewhere on your system for your projects. It’s best to use lowercase letters and underscores for spaces in file and folder names, because Python uses these naming conventions.

Open Sublime Text, and save an empty Python file (**File** ▶ **Save As**) called `hello_world.py` in your `python_work` folder. The extension `.py` tells Sublime Text that the code in your file is written in Python, which tells it how to run the program and highlight the text in a helpful way.

After you’ve saved your file, enter the following line in the text editor:

`hello_world.py`

```
print("Hello Python world!")
```

If the `python` command works on your system, you can run your program by selecting **Tools** ▶ **Build** in the menu or by pressing **CTRL-B** (⌘-B on macOS). If you had to configure Sublime Text in the previous section, select **Tools** ▶ **Build System** and then select **Python 3**. From now on you’ll be able to select **Tools** ▶ **Build** or just press **CTRL-B** (or ⌘-B) to run your programs.

A terminal screen should appear at the bottom of the Sublime Text window, showing the following output:

```
Hello Python world!
[Finished in 0.1s]
```

If you don’t see this output, something might have gone wrong in the program. Check every character on the line you entered. Did you accidentally capitalize `print`? Did you forget one or both of the quotation marks or parentheses? Programming languages expect very specific syntax, and if you don’t provide that, you’ll get errors. If you can’t get the program to run, see the suggestions in the next section.

Troubleshooting

If you can't get *hello_world.py* to run, here are a few remedies you can try that are also good general solutions for any programming problem:

- When a program contains a significant error, Python displays a *traceback*, which is an error report. Python looks through the file and tries to identify the problem. Check the traceback; it might give you a clue as to what issue is preventing the program from running.
- Step away from your computer, take a short break, and then try again. Remember that syntax is very important in programming, so even a missing colon, a mismatched quotation mark, or mismatched parentheses can prevent a program from running properly. Reread the relevant parts of this chapter, look over your code, and try to find the mistake.
- Start over again. You probably don't need to uninstall any software, but it might make sense to delete your *hello_world.py* file and re-create it from scratch.
- Ask someone else to follow the steps in this chapter, on your computer or a different one, and watch what they do carefully. You might have missed one small step that someone else happens to catch.
- Find someone who knows Python and ask them to help you get set up. If you ask around, you might find that you unexpectedly know someone who uses Python.
- The setup instructions in this chapter are also available through the book's companion website at <https://nostarch.com/pythoncrashcourse2e/>. The online version of these instructions might work better for you because you can simply cut and paste code.
- Ask for help online. Appendix C provides a number of resources, such as forums and live chat sites, where you can ask for solutions from people who've already worked through the issue you're currently facing.

Never worry that you're bothering experienced programmers. Every programmer has been stuck at some point, and most programmers are happy to help you set up your system correctly. As long as you can state clearly what you're trying to do, what you've already tried, and the results you're getting, there's a good chance someone will be able to help you. As mentioned in the Introduction, the Python community is very friendly and welcoming to beginners.

Python should run well on any modern computer. Early setup issues can be frustrating, but they're well worth sorting out. Once you get *hello_world.py* running, you can start to learn Python, and your programming work will become more interesting and satisfying.

Running Python Programs from a Terminal

Most of the programs you write in your text editor you'll run directly from the editor. But sometimes it's useful to run programs from a terminal instead. For example, you might want to run an existing program without opening it for editing.

You can do this on any system with Python installed if you know how to access the directory where the program file is stored. To try this, make sure you've saved the *hello_world.py* file in the *python_work* folder on your desktop.

On Windows

You can use the terminal command `cd`, for *change directory*, to navigate through your filesystem in a command window. The command `dir`, for *directory*, shows you all the files that exist in the current directory.

Open a new terminal window and enter the following commands to run *hello_world.py*:

```
❶ C:\> cd Desktop\python_work
❷ C:\Desktop\python_work> dir
    hello_world.py
❸ C:\Desktop\python_work> python hello_world.py
Hello Python world!
```

At ❶ you use the `cd` command to navigate to the *python_work* folder, which is in the *Desktop* folder. Next, you use the `dir` command to make sure *hello_world.py* is in this folder ❷. Then you run the file using the command `python hello_world.py` ❸.

Most of your programs will run fine directly from your editor. But as your work becomes more complex, you'll want to run some of your programs from a terminal.

On macOS and Linux

Running a Python program from a terminal session is the same on Linux and macOS. You can use the terminal command `cd`, for *change directory*, to navigate through your filesystem in a terminal session. The command `ls`, for *list*, shows you all the nonhidden files that exist in the current directory.

Open a new terminal window and enter the following commands to run *hello_world.py*:

```
❶ ~$ cd Desktop/python_work/
❷ ~/Desktop/python_work$ ls
    hello_world.py
❸ ~/Desktop/python_work$ python hello_world.py
Hello Python world!
```

At ❶ you use the `cd` command to navigate to the `python_work` folder, which is in the `Desktop` folder. Next, you use the `ls` command to make sure `hello_world.py` is in this folder ❷. Then you run the file using the command `python hello_world.py` ❸.

It's that simple. You just use the `python` (or `python3`) command to run Python programs.

TRY IT YOURSELF

The exercises in this chapter are exploratory in nature. Starting in Chapter 2, the challenges you'll solve will be based on what you've learned.

1-1. `python.org`: Explore the Python home page (<https://python.org/>) to find topics that interest you. As you become familiar with Python, different parts of the site will be more useful to you.

1-2. Hello World Typos: Open the `hello_world.py` file you just created. Make a typo somewhere in the line and run the program again. Can you make a typo that generates an error? Can you make sense of the error message? Can you make a typo that doesn't generate an error? Why do you think it didn't make an error?

1-3. Infinite Skills: If you had infinite programming skills, what would you build? You're about to learn how to program. If you have an end goal in mind, you'll have an immediate use for your new skills; now is a great time to draft descriptions of what you want to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. Take a few minutes now to describe three programs you want to create.

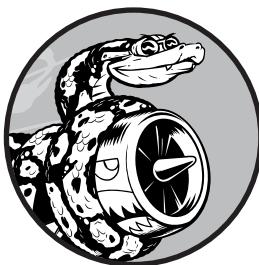
Summary

In this chapter, you learned a bit about Python in general, and you installed Python on your system if it wasn't already there. You also installed a text editor to make it easier to write Python code. You ran snippets of Python code in a terminal session, and you ran your first program, `hello_world.py`. You probably learned a bit about troubleshooting as well.

In the next chapter, you'll learn about the different kinds of data you can work with in your Python programs, and you'll use variables as well.

2

VARIABLES AND SIMPLE DATA TYPES



In this chapter you'll learn about the different kinds of data you can work with in your Python programs. You'll also learn how to use variables to represent data in your programs.

What Really Happens When You Run `hello_world.py`

Let's take a closer look at what Python does when you run `hello_world.py`. As it turns out, Python does a fair amount of work, even when it runs a simple program:

`hello_world.py`

```
print("Hello Python world!")
```

When you run this code, you should see this output:

`Hello Python world!`

When you run the file *hello_world.py*, the ending *.py* indicates that the file is a Python program. Your editor then runs the file through the *Python interpreter*, which reads through the program and determines what each word in the program means. For example, when the interpreter sees the word `print` followed by parentheses, it prints to the screen whatever is inside the parentheses.

As you write your programs, your editor highlights different parts of your program in different ways. For example, it recognizes that `print()` is the name of a function and displays that word in one color. It recognizes that "Hello Python world!" is not Python code and displays that phrase in a different color. This feature is called *syntax highlighting* and is quite useful as you start to write your own programs.

Variables

Let's try using a variable in *hello_world.py*. Add a new line at the beginning of the file, and modify the second line:

hello_world.py

```
message = "Hello Python world!"
print(message)
```

Run this program to see what happens. You should see the same output you saw previously:

Hello Python world!

We've added a *variable* named `message`. Every variable is connected to a *value*, which is the information associated with that variable. In this case the value is the "Hello Python world!" text.

Adding a variable makes a little more work for the Python interpreter. When it processes the first line, it associates the variable `message` with the "Hello Python world!" text. When it reaches the second line, it prints the value associated with `message` to the screen.

Let's expand on this program by modifying *hello_world.py* to print a second message. Add a blank line to *hello_world.py*, and then add two new lines of code:

```
message = "Hello Python world!"
print(message)

message = "Hello Python Crash Course world!"
print(message)
```

Now when you run *hello_world.py*, you should see two lines of output:

Hello Python world!
Hello Python Crash Course world!

You can change the value of a variable in your program at any time, and Python will always keep track of its current value.

Naming and Using Variables

When you're using variables in Python, you need to adhere to a few rules and guidelines. Breaking some of these rules will cause errors; other guidelines just help you write code that's easier to read and understand. Be sure to keep the following variable rules in mind:

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable *message_1* but not *1_message*.
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, *greeting_message* works, but *greeting message* will cause errors.
- Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular programmatic purpose, such as the word *print*. (See "Python Keywords and Built-in Functions" on page 471.)
- Variable names should be short but descriptive. For example, *name* is better than *n*, *student_name* is better than *s_n*, and *name_length* is better than *length_of_persons_name*.
- Be careful when using the lowercase letter *l* and the uppercase letter *O* because they could be confused with the numbers *1* and *0*.

It can take some practice to learn how to create good variable names, especially as your programs become more interesting and complicated. As you write more programs and start to read through other people's code, you'll get better at coming up with meaningful names.

NOTE

The Python variables you're using at this point should be lowercase. You won't get errors if you use uppercase letters, but uppercase letters in variable names have special meanings that we'll discuss in later chapters.

Avoiding Name Errors When Using Variables

Every programmer makes mistakes, and most make mistakes every day. Although good programmers might create errors, they also know how to respond to those errors efficiently. Let's look at an error you're likely to make early on and learn how to fix it.

We'll write some code that generates an error on purpose. Enter the following code, including the misspelled word *mesage* shown in bold:

```
message = "Hello Python Crash Course reader!"  
print(mesage)
```

When an error occurs in your program, the Python interpreter does its best to help you figure out where the problem is. The interpreter provides a traceback when a program cannot run successfully. A *traceback* is a record of where the interpreter ran into trouble when trying to execute your code. Here's an example of the traceback that Python provides after you've accidentally misspelled a variable's name:

```
Traceback (most recent call last):
❶  File "hello_world.py", line 2, in <module>
❷      print(mesage)
❸ NameError: name 'mesage' is not defined
```

The output at ❶ reports that an error occurs in line 2 of the file *hello_world.py*. The interpreter shows this line ❷ to help us spot the error quickly and tells us what kind of error it found ❸. In this case it found a *name error* and reports that the variable being printed, *mesage*, has not been defined. Python can't identify the variable name provided. A name error usually means we either forgot to set a variable's value before using it, or we made a spelling mistake when entering the variable's name.

Of course, in this example we omitted the letter *s* in the variable name *message* in the second line. The Python interpreter doesn't spellcheck your code, but it does ensure that variable names are spelled consistently. For example, watch what happens when we spell *message* incorrectly in another place in the code as well:

```
mesage = "Hello Python Crash Course reader!"
print(mesage)
```

In this case, the program runs successfully!

```
Hello Python Crash Course reader!
```

Programming languages are strict, but they disregard good and bad spelling. As a result, you don't need to consider English spelling and grammar rules when you're trying to create variable names and writing code.

Many programming errors are simple, single-character typos in one line of a program. If you're spending a long time searching for one of these errors, know that you're in good company. Many experienced and talented programmers spend hours hunting down these kinds of tiny errors. Try to laugh about it and move on, knowing it will happen frequently throughout your programming life.

Variables Are Labels

Variables are often described as boxes you can store values in. This idea can be helpful the first few times you use a variable, but it isn't an accurate way to describe how variables are represented internally in Python. It's much better to think of variables as labels that you can assign to values. You can also say that a variable references a certain value.

This distinction probably won't matter much in your initial programs, but it's worth learning earlier rather than later. At some point, you'll see unexpected behavior from a variable, and an accurate understanding of how variables work will help you identify what's happening in your code.

NOTE

The best way to understand new programming concepts is to try using them in your programs. If you get stuck while working on an exercise in this book, try doing something else for a while. If you're still stuck, review the relevant part of that chapter. If you still need help, see the suggestions in Appendix C.

TRY IT YOURSELF

Write a separate program to accomplish each of these exercises. Save each program with a filename that follows standard Python conventions, using lowercase letters and underscores, such as `simple_message.py` and `simple_messages.py`.

2-1. Simple Message: Assign a message to a variable, and then print that message.

2-2. Simple Messages: Assign a message to a variable, and print that message. Then change the value of the variable to a new message, and print the new message.

Strings

Because most programs define and gather some sort of data, and then do something useful with it, it helps to classify different types of data. The first data type we'll look at is the string. Strings are quite simple at first glance, but you can use them in many different ways.

A *string* is a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings like this:

```
"This is a string."  
'This is also a string.'
```

This flexibility allows you to use quotes and apostrophes within your strings:

```
'I told my friend, "Python is my favorite language!"'  
"The language 'Python' is named after Monty Python, not the snake."  
"One of Python's strengths is its diverse and supportive community."
```

Let's explore some of the ways you can use strings.

Changing Case in a String with Methods

One of the simplest tasks you can do with strings is change the case of the words in a string. Look at the following code, and try to determine what's happening:

name.py

```
name = "ada lovelace"
print(name.title())
```

Save this file as *name.py*, and then run it. You should see this output:

Ada Lovelace

In this example, the variable `name` refers to the lowercase string "ada lovelace". The method `title()` appears after the variable in the `print()` call. A *method* is an action that Python can perform on a piece of data. The dot (.) after `name` in `name.title()` tells Python to make the `title()` method act on the variable `name`. Every method is followed by a set of parentheses, because methods often need additional information to do their work. That information is provided inside the parentheses. The `title()` function doesn't need any additional information, so its parentheses are empty.

The `title()` method changes each word to title case, where each word begins with a capital letter. This is useful because you'll often want to think of a name as a piece of information. For example, you might want your program to recognize the input values Ada, ADA, and ada as the same name, and display all of them as Ada.

Several other useful methods are available for dealing with case as well. For example, you can change a string to all uppercase or all lowercase letters like this:

```
name = "Ada Lovelace"
print(name.upper())
print(name.lower())
```

This will display the following:

ADA LOVELACE
ada lovelace

The `lower()` method is particularly useful for storing data. Many times you won't want to trust the capitalization that your users provide, so you'll convert strings to lowercase before storing them. Then when you want to display the information, you'll use the case that makes the most sense for each string.

Using Variables in Strings

In some situations, you'll want to use a variable's value inside a string. For example, you might want two variables to represent a first name and a last name respectively, and then want to combine those values to display someone's full name:

```
full_name.py
first_name = "ada"
last_name = "lovelace"
❶ full_name = f"{first_name} {last_name}"
print(full_name)
```

To insert a variable's value into a string, place the letter `f` immediately before the opening quotation mark ❶. Put braces around the name or names of any variable you want to use inside the string. Python will replace each variable with its value when the string is displayed.

These strings are called *f-strings*. The `f` is for *format*, because Python formats the string by replacing the name of any variable in braces with its value. The output from the previous code is:

```
ada lovelace
```

You can do a lot with f-strings. For example, you can use f-strings to compose complete messages using the information associated with a variable, as shown here:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
❶ print(f"Hello, {full_name.title()}!")
```

The full name is used in a sentence that greets the user ❶, and the `title()` method changes the name to title case. This code returns a simple but nicely formatted greeting:

```
Hello, Ada Lovelace!
```

You can also use f-strings to compose a message, and then assign the entire message to a variable:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
❶ message = f"Hello, {full_name.title()}!"
❷ print(message)
```

This code displays the message `Hello, Ada Lovelace!` as well, but by assigning the message to a variable ❶ we make the final `print()` call much simpler ❷.

NOTE

F-strings were first introduced in Python 3.6. If you’re using Python 3.5 or earlier, you’ll need to use the `format()` method rather than this `f` syntax. To use `format()`, list the variables you want to use in the string inside the parentheses following `format`. Each variable is referred to by a set of braces; the braces will be filled by the values listed in parentheses in the order provided:

```
full_name = "{} {}".format(first_name, last_name)
```

Adding Whitespace to Strings with Tabs or Newlines

In programming, *whitespace* refers to any nonprinting character, such as spaces, tabs, and end-of-line symbols. You can use whitespace to organize your output so it’s easier for users to read.

To add a tab to your text, use the character combination `\t` as shown at ❶:

```
>>> print("Python")
Python
❶ >>> print("\tPython")
Python
```

To add a newline in a string, use the character combination `\n`:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

You can also combine tabs and newlines in a single string. The string `"\n\t"` tells Python to move to a new line, and start the next line with a tab. The following example shows how you can use a one-line string to generate four lines of output:

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
```

Languages:
Python
C
JavaScript

Newlines and tabs will be very useful in the next two chapters when you start to produce many lines of output from just a few lines of code.

Stripping Whitespace

Extra whitespace can be confusing in your programs. To programmers `'python'` and `'python '` look pretty much the same. But to a program, they are two different strings. Python detects the extra space in `'python '` and considers it significant unless you tell it otherwise.

It's important to think about whitespace, because often you'll want to compare two strings to determine whether they are the same. For example, one important instance might involve checking people's usernames when they log in to a website. Extra whitespace can be confusing in much simpler situations as well. Fortunately, Python makes it easy to eliminate extraneous whitespace from data that people enter.

Python can look for extra whitespace on the right and left sides of a string. To ensure that no whitespace exists at the right end of a string, use the `rstrip()` method.

```
❶ >>> favorite_language = 'python '
❷ >>> favorite_language
'python '
❸ >>> favorite_language.rstrip()
'python'
❹ >>> favorite_language
'python '
```

The value associated with `favorite_language` at ❶ contains extra whitespace at the end of the string. When you ask Python for this value in a terminal session, you can see the space at the end of the value ❷. When the `rstrip()` method acts on the variable `favorite_language` at ❸, this extra space is removed. However, it is only removed temporarily. If you ask for the value of `favorite_language` again, you can see that the string looks the same as when it was entered, including the extra whitespace ❹.

To remove the whitespace from the string permanently, you have to associate the stripped value with the variable name:

```
>>> favorite_language = 'python '
❶ >>> favorite_language = favorite_language.rstrip()
>>> favorite_language
'python'
```

To remove the whitespace from the string, you strip the whitespace from the right side of the string and then associate this new value with the original variable, as shown at ❶. Changing a variable's value is done often in programming. This is how a variable's value can be updated as a program is executed or in response to user input.

You can also strip whitespace from the left side of a string using the `lstrip()` method, or from both sides at once using `strip()`:

```
❶ >>> favorite_language = ' python '
❷ >>> favorite_language.rstrip()
' python'
❸ >>> favorite_language.lstrip()
' python'
❹ >>> favorite_language.strip()
'python'
```

In this example, we start with a value that has whitespace at the beginning and the end ❶. We then remove the extra space from the right side at ❷, from the left side at ❸, and from both sides at ❹. Experimenting with these stripping functions can help you become familiar with manipulating strings. In the real world, these stripping functions are used most often to clean up user input before it's stored in a program.

Avoiding Syntax Errors with Strings

One kind of error that you might see with some regularity is a syntax error. A *syntax error* occurs when Python doesn't recognize a section of your program as valid Python code. For example, if you use an apostrophe within single quotes, you'll produce an error. This happens because Python interprets everything between the first single quote and the apostrophe as a string. It then tries to interpret the rest of the text as Python code, which causes errors.

Here's how to use single and double quotes correctly. Save this program as *apostrophe.py* and then run it:

apostrophe.py

```
message = "One of Python's strengths is its diverse community."  
print(message)
```

The apostrophe appears inside a set of double quotes, so the Python interpreter has no trouble reading the string correctly:

One of Python's strengths is its diverse community.

However, if you use single quotes, Python can't identify where the string should end:

```
message = 'One of Python's strengths is its diverse community.'  
print(message)
```

You'll see the following output:

```
File "apostrophe.py", line 1  
    message = 'One of Python's strengths is its diverse community.'  
               ^❶  
SyntaxError: invalid syntax
```

In the output you can see that the error occurs at ❶ right after the second single quote. This syntax error indicates that the interpreter doesn't recognize something in the code as valid Python code. Errors can come from a variety of sources, and I'll point out some common ones as they arise. You might see syntax errors often as you learn to write proper Python code. Syntax errors are also the least specific kind of error, so they can be difficult and frustrating to identify and correct. If you get stuck on a particularly stubborn error, see the suggestions in Appendix C.

NOTE

Your editor's syntax highlighting feature should help you spot some syntax errors quickly as you write your programs. If you see Python code highlighted as if it's English or English highlighted as if it's Python code, you probably have a mismatched quotation mark somewhere in your file.

TRY IT YOURSELF

Save each of the following exercises as a separate file with a name like `name_cases.py`. If you get stuck, take a break or see the suggestions in Appendix C.

2-3. Personal Message: Use a variable to represent a person's name, and print a message to that person. Your message should be simple, such as, "Hello Eric, would you like to learn some Python today?"

2-4. Name Cases: Use a variable to represent a person's name, and then print that person's name in lowercase, uppercase, and title case.

2-5. Famous Quote: Find a quote from a famous person you admire. Print the quote and the name of its author. Your output should look something like the following, including the quotation marks:

Albert Einstein once said, "A person who never made a mistake never tried anything new."

2-6. Famous Quote 2: Repeat Exercise 2-5, but this time, represent the famous person's name using a variable called `famous_person`. Then compose your message and represent it with a new variable called `message`. Print your message.

2-7. Stripping Names: Use a variable to represent a person's name, and include some whitespace characters at the beginning and end of the name. Make sure you use each character combination, "\t" and "\n", at least once.

Print the name once, so the whitespace around the name is displayed. Then print the name using each of the three stripping functions, `lstrip()`, `rstrip()`, and `strip()`.

Numbers

Numbers are used quite often in programming to keep score in games, represent data in visualizations, store information in web applications, and so on. Python treats numbers in several different ways, depending on how they're being used. Let's first look at how Python manages integers, because they're the simplest to work with.

Integers

You can add (+), subtract (-), multiply (*), and divide (/) integers in Python.

```
>>> 2 + 3  
5  
>>> 3 - 2  
1  
>>> 2 * 3  
6  
>>> 3 / 2  
1.5
```

In a terminal session, Python simply returns the result of the operation. Python uses two multiplication symbols to represent exponents:

```
>>> 3 ** 2  
9  
>>> 3 ** 3  
27  
>>> 10 ** 6  
1000000
```

Python supports the order of operations too, so you can use multiple operations in one expression. You can also use parentheses to modify the order of operations so Python can evaluate your expression in the order you specify. For example:

```
>>> 2 + 3*4  
14  
>>> (2 + 3) * 4  
20
```

The spacing in these examples has no effect on how Python evaluates the expressions; it simply helps you more quickly spot the operations that have priority when you're reading through the code.

Floats

Python calls any number with a decimal point a *float*. This term is used in most programming languages, and it refers to the fact that a decimal point can appear at any position in a number. Every programming language must be carefully designed to properly manage decimal numbers so numbers behave appropriately no matter where the decimal point appears.

For the most part, you can use decimals without worrying about how they behave. Simply enter the numbers you want to use, and Python will most likely do what you expect:

```
>>> 0.1 + 0.1  
0.2  
>>> 0.2 + 0.2  
0.4  
>>> 2 * 0.1  
0.2  
>>> 2 * 0.2  
0.4
```

But be aware that you can sometimes get an arbitrary number of decimal places in your answer:

```
>>> 0.2 + 0.1  
0.3000000000000004  
>>> 3 * 0.1  
0.3000000000000004
```

This happens in all languages and is of little concern. Python tries to find a way to represent the result as precisely as possible, which is sometimes difficult given how computers have to represent numbers internally. Just ignore the extra decimal places for now; you'll learn ways to deal with the extra places when you need to in the projects in Part II.

Integers and Floats

When you divide any two numbers, even if they are integers that result in a whole number, you'll always get a float:

```
>>> 4/2  
2.0
```

If you mix an integer and a float in any other operation, you'll get a float as well:

```
>>> 1 + 2.0  
3.0  
>>> 2 * 3.0  
6.0  
>>> 3.0 ** 2  
9.0
```

Python defaults to a float in any operation that uses a float, even if the output is a whole number.

Underscores in Numbers

When you’re writing long numbers, you can group digits using underscores to make large numbers more readable:

```
>>> universe_age = 14_000_000_000
```

When you print a number that was defined using underscores, Python prints only the digits:

```
>>> print(universe_age)  
14000000000
```

Python ignores the underscores when storing these kinds of values. Even if you don’t group the digits in threes, the value will still be unaffected. To Python, 1000 is the same as 1_000, which is the same as 10_00. This feature works for integers and floats, but it’s only available in Python 3.6 and later.

Multiple Assignment

You can assign values to more than one variable using just a single line. This can help shorten your programs and make them easier to read; you’ll use this technique most often when initializing a set of numbers.

For example, here’s how you can initialize the variables `x`, `y`, and `z` to zero:

```
>>> x, y, z = 0, 0, 0
```

You need to separate the variable names with commas, and do the same with the values, and Python will assign each value to its respectively positioned variable. As long as the number of values matches the number of variables, Python will match them up correctly.

Constants

A *constant* is like a variable whose value stays the same throughout the life of a program. Python doesn’t have built-in constant types, but Python programmers use all capital letters to indicate a variable should be treated as a constant and never be changed:

```
MAX_CONNECTIONS = 5000
```

When you want to treat a variable as a constant in your code, make the name of the variable all capital letters.

TRY IT YOURSELF

2-8. Number Eight: Write addition, subtraction, multiplication, and division operations that each result in the number 8. Be sure to enclose your operations in `print()` calls to see the results. You should create four lines that look like this:

```
print(5+3)
```

Your output should simply be four lines with the number 8 appearing once on each line.

2-9. Favorite Number: Use a variable to represent your favorite number. Then, using that variable, create a message that reveals your favorite number. Print that message.

Comments

Comments are an extremely useful feature in most programming languages. Everything you've written in your programs so far is Python code. As your programs become longer and more complicated, you should add notes within your programs that describe your overall approach to the problem you're solving. A *comment* allows you to write notes in English within your programs.

How Do You Write Comments?

In Python, the hash mark (#) indicates a comment. Anything following a hash mark in your code is ignored by the Python interpreter. For example:

`comment.py`

```
# Say hello to everyone.  
print("Hello Python people!")
```

Python ignores the first line and executes the second line.

Hello Python people!

What Kind of Comments Should You Write?

The main reason to write comments is to explain what your code is supposed to do and how you are making it work. When you're in the middle of working on a project, you understand how all of the pieces fit together. But when you return to a project after some time away, you'll likely have forgotten

some of the details. You can always study your code for a while and figure out how segments were supposed to work, but writing good comments can save you time by summarizing your overall approach in clear English.

If you want to become a professional programmer or collaborate with other programmers, you should write meaningful comments. Today, most software is written collaboratively, whether by a group of employees at one company or a group of people working together on an open source project. Skilled programmers expect to see comments in code, so it's best to start adding descriptive comments to your programs now. Writing clear, concise comments in your code is one of the most beneficial habits you can form as a new programmer.

When you're determining whether to write a comment, ask yourself if you had to consider several approaches before coming up with a reasonable way to make something work; if so, write a comment about your solution. It's much easier to delete extra comments later on than it is to go back and write comments for a sparsely commented program. From now on, I'll use comments in examples throughout this book to help explain sections of code.

TRY IT YOURSELF

2-10. Adding Comments: Choose two of the programs you've written, and add at least one comment to each. If you don't have anything specific to write because your programs are too simple at this point, just add your name and the current date at the top of each program file. Then write one sentence describing what the program does.

The Zen of Python

Experienced Python programmers will encourage you to avoid complexity and aim for simplicity whenever possible. The Python community's philosophy is contained in "The Zen of Python" by Tim Peters. You can access this brief set of principles for writing good Python code by entering `import this` into your interpreter. I won't reproduce the entire "Zen of Python" here, but I'll share a few lines to help you understand why they should be important to you as a beginning Python programmer.

```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
```

Python programmers embrace the notion that code can be beautiful and elegant. In programming, people solve problems. Programmers have always respected well-designed, efficient, and even beautiful solutions to

problems. As you learn more about Python and use it to write more code, someone might look over your shoulder one day and say, “Wow, that’s some beautiful code!”

Simple is better than complex.

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

Complex is better than complicated.

Real life is messy, and sometimes a simple solution to a problem is unattainable. In that case, use the simplest solution that works.

Readability counts.

Even when your code is complex, aim to make it readable. When you’re working on a project that involves complex coding, focus on writing informative comments for that code.

There should be one-- and preferably only one --obvious way to do it.

If two Python programmers are asked to solve the same problem, they should come up with fairly compatible solutions. This is not to say there’s no room for creativity in programming. On the contrary! But much of programming consists of using small, common approaches to simple situations within a larger, more creative project. The nuts and bolts of your programs should make sense to other Python programmers.

Now is better than never.

You could spend the rest of your life learning all the intricacies of Python and of programming in general, but then you’d never complete any projects. Don’t try to write perfect code; write code that works, and then decide whether to improve your code for that project or move on to something new.

As you continue to the next chapter and start digging into more involved topics, try to keep this philosophy of simplicity and clarity in mind. Experienced programmers will respect your code more and will be happy to give you feedback and collaborate with you on interesting projects.

TRY IT YOURSELF

2-11. Zen of Python: Enter `import this` into a Python terminal session and skim through the additional principles.

Summary

In this chapter you learned to work with variables. You learned to use descriptive variable names and how to resolve name errors and syntax errors when they arise. You learned what strings are and how to display strings using lowercase, uppercase, and title case. You started using whitespace to organize output neatly, and you learned to strip unneeded whitespace from different parts of a string. You started working with integers and floats, and learned some of the ways you can work with numerical data. You also learned to write explanatory comments to make your code easier for you and others to read. Finally, you read about the philosophy of keeping your code as simple as possible, whenever possible.

In Chapter 3 you'll learn to store collections of information in data structures called *lists*. You'll learn to work through a list, manipulating any information in that list.

3

INTRODUCING LISTS



In this chapter and the next you'll learn what lists are and how to start working with the elements in a list. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.

What Is a List?

A *list* is a collection of items in a particular order. You can make a list that includes the letters of the alphabet, the digits from 0–9, or the names of all the people in your family. You can put anything you want into a list, and

the items in your list don't have to be related in any particular way. Because a list usually contains more than one element, it's a good idea to make the name of your list plural, such as `letters`, `digits`, or `names`.

In Python, square brackets (`[]`) indicate a list, and individual elements in the list are separated by commas. Here's a simple example of a list that contains a few kinds of bicycles:

`bicycles.py`

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

If you ask Python to print a list, Python returns its representation of the list, including the square brackets:

```
['trek', 'cannondale', 'redline', 'specialized']
```

Because this isn't the output you want your users to see, let's learn how to access the individual items in a list.

Accessing Elements in a List

Lists are ordered collections, so you can access any element in a list by telling Python the position, or *index*, of the item desired. To access an element in a list, write the name of the list followed by the index of the item enclosed in square brackets.

For example, let's pull out the first bicycle in the list `bicycles`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ print(bicycles[0])
```

The syntax for this is shown at ❶. When we ask for a single item from a list, Python returns just that element without square brackets:

```
trek
```

This is the result you want your users to see—clean, neatly formatted output.

You can also use the string methods from Chapter 2 on any element in this list. For example, you can format the element `'trek'` more neatly by using the `title()` method:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0].title())
```

This example produces the same output as the preceding example except `'Trek'` is capitalized.

Index Positions Start at 0, Not 1

Python considers the first item in a list to be at position 0, not position 1. This is true of most programming languages, and the reason has to do with how the list operations are implemented at a lower level. If you're receiving unexpected results, determine whether you are making a simple off-by-one error.

The second item in a list has an index of 1. Using this counting system, you can get any element you want from a list by subtracting one from its position in the list. For instance, to access the fourth item in a list, you request the item at index 3.

The following asks for the bicycles at index 1 and index 3:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[1])
print(bicycles[3])
```

This code returns the second and fourth bicycles in the list:

```
cannondale
specialized
```

Python has a special syntax for accessing the last element in a list. By asking for the item at index -1, Python always returns the last item in the list:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[-1])
```

This code returns the value 'specialized'. This syntax is quite useful, because you'll often want to access the last items in a list without knowing exactly how long the list is. This convention extends to other negative index values as well. The index -2 returns the second item from the end of the list, the index -3 returns the third item from the end, and so forth.

Using Individual Values from a List

You can use individual values from a list just as you would any other variable. For example, you can use f-strings to create a message based on a value from a list.

Let's try pulling the first bicycle from the list and composing a message using that value.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ message = f"My first bicycle was a {bicycles[0].title()}."  
print(message)
```

At ❶, we build a sentence using the value at `bicycles[0]` and assign it to the variable `message`. The output is a simple sentence about the first bicycle in the list:

My first bicycle was a Trek.

TRY IT YOURSELF

Try these short programs to get some firsthand experience with Python’s lists. You might want to create a new folder for each chapter’s exercises to keep them organized.

3-1. Names: Store the names of a few of your friends in a list called `names`. Print each person’s name by accessing each element in the list, one at a time.

3-2. Greetings: Start with the list you used in Exercise 3-1, but instead of just printing each person’s name, print a message to them. The text of each message should be the same, but each message should be personalized with the person’s name.

3-3. Your Own List: Think of your favorite mode of transportation, such as a motorcycle or a car, and make a list that stores several examples. Use your list to print a series of statements about these items, such as “I would like to own a Honda motorcycle.”

Changing, Adding, and Removing Elements

Most lists you create will be dynamic, meaning you’ll build a list and then add and remove elements from it as your program runs its course. For example, you might create a game in which a player has to shoot aliens out of the sky. You could store the initial set of aliens in a list and then remove an alien from the list each time one is shot down. Each time a new alien appears on the screen, you add it to the list. Your list of aliens will increase and decrease in length throughout the course of the game.

Modifying Elements in a List

The syntax for modifying an element is similar to the syntax for accessing an element in a list. To change an element, use the name of the list followed by the index of the element you want to change, and then provide the new value you want that item to have.

For example, let's say we have a list of motorcycles, and the first item in the list is 'honda'. How would we change the value of this first item?

```
motorcycles.py ❶ motorcycles = ['honda', 'yamaha', 'suzuki']
               print(motorcycles)

❷ motorcycles[0] = 'ducati'
               print(motorcycles)
```

The code at ❶ defines the original list, with 'honda' as the first element. The code at ❷ changes the value of the first item to 'ducati'. The output shows that the first item has indeed been changed, and the rest of the list stays the same:

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

You can change the value of any item in a list, not just the first item.

Adding Elements to a List

You might want to add a new element to a list for many reasons. For example, you might want to make new aliens appear in a game, add new data to a visualization, or add new registered users to a website you've built. Python provides several ways to add new data to existing lists.

Appending Elements to the End of a List

The simplest way to add a new element to a list is to *append* the item to the list. When you append an item to a list, the new element is added to the end of the list. Using the same list we had in the previous example, we'll add the new element 'ducati' to the end of the list:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
```

```
❶ motorcycles.append('ducati')
print(motorcycles)
```

The `append()` method at ❶ adds 'ducati' to the end of the list without affecting any of the other elements in the list:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha', 'suzuki', 'ducati']
```

The `append()` method makes it easy to build lists dynamically. For example, you can start with an empty list and then add items to the list using a series of `append()` calls. Using an empty list, let's add the elements '`honda`', '`yamaha`', and '`suzuki`' to the list:

```
motorcycles = []  
  
motorcycles.append('honda')  
motorcycles.append('yamaha')  
motorcycles.append('suzuki')  
  
print(motorcycles)
```

The resulting list looks exactly the same as the lists in the previous examples:

```
['honda', 'yamaha', 'suzuki']
```

Building lists this way is very common, because you often won't know the data your users want to store in a program until after the program is running. To put your users in control, start by defining an empty list that will hold the users' values. Then append each new value provided to the list you just created.

Inserting Elements into a List

You can add a new element at any position in your list by using the `insert()` method. You do this by specifying the index of the new element and the value of the new item.

```
motorcycles = ['honda', 'yamaha', 'suzuki']  
  
❶ motorcycles.insert(0, 'ducati')  
print(motorcycles)
```

In this example, the code at ❶ inserts the value '`ducati`' at the beginning of the list. The `insert()` method opens a space at position 0 and stores the value '`ducati`' at that location. This operation shifts every other value in the list one position to the right:

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

Removing Elements from a List

Often, you'll want to remove an item or a set of items from a list. For example, when a player shoots down an alien from the sky, you'll most likely want to remove it from the list of active aliens. Or when a user

decides to cancel their account on a web application you created, you'll want to remove that user from the list of active users. You can remove an item according to its position in the list or according to its value.

Removing an Item Using the `del` Statement

If you know the position of the item you want to remove from a list, you can use the `del` statement.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
```

```
❶ del motorcycles[0]
print(motorcycles)
```

The code at ❶ uses `del` to remove the first item, 'honda', from the list of motorcycles:

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

You can remove an item from any position in a list using the `del` statement if you know its index. For example, here's how to remove the second item, 'yamaha', in the list:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
```

```
del motorcycles[1]
print(motorcycles)
```

The second motorcycle is deleted from the list:

```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
```

In both examples, you can no longer access the value that was removed from the list after the `del` statement is used.

Removing an Item Using the `pop()` Method

Sometimes you'll want to use the value of an item after you remove it from a list. For example, you might want to get the *x* and *y* position of an alien that was just shot down, so you can draw an explosion at that position. In a web application, you might want to remove a user from a list of active members and then add that user to a list of inactive members.

The `pop()` method removes the last item in a list, but it lets you work with that item after removing it. The term *pop* comes from thinking of a list as a stack of items and popping one item off the top of the stack. In this analogy, the top of a stack corresponds to the end of a list.

Let's pop a motorcycle from the list of motorcycles:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

❷ popped_motorcycle = motorcycles.pop()
❸ print(motorcycles)
❹ print(popped_motorcycle)
```

We start by defining and printing the list `motorcycles` at ❶. At ❷ we pop a value from the list and store that value in the variable `popped_motorcycle`. We print the list at ❸ to show that a value has been removed from the list. Then we print the popped value at ❹ to prove that we still have access to the value that was removed.

The output shows that the value 'suzuki' was removed from the end of the list and is now assigned to the variable `popped_motorcycle`:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

How might this `pop()` method be useful? Imagine that the motorcycles in the list are stored in chronological order according to when we owned them. If this is the case, we can use the `pop()` method to print a statement about the last motorcycle we bought:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

last_owned = motorcycles.pop()
print(f"The last motorcycle I owned was a {last_owned.title()}.")
```

The output is a simple sentence about the most recent motorcycle we owned:

The last motorcycle I owned was a Suzuki.

Popping Items from any Position in a List

You can use `pop()` to remove an item from any position in a list by including the index of the item you want to remove in parentheses.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

❶ first_owned = motorcycles.pop(0)
❷ print(f"The first motorcycle I owned was a {first_owned.title()}.")
```

We start by popping the first motorcycle in the list at ❶, and then we print a message about that motorcycle at ❷. The output is a simple sentence describing the first motorcycle I ever owned:

```
The first motorcycle I owned was a Honda.
```

Remember that each time you use `pop()`, the item you work with is no longer stored in the list.

If you're unsure whether to use the `del` statement or the `pop()` method, here's a simple way to decide: when you want to delete an item from a list and not use that item in any way, use the `del` statement; if you want to use an item as you remove it, use the `pop()` method.

Removing an Item by Value

Sometimes you won't know the position of the value you want to remove from a list. If you only know the value of the item you want to remove, you can use the `remove()` method.

For example, let's say we want to remove the value 'ducati' from the list of motorcycles.

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)
```

```
❶ motorcycles.remove('ducati')
print(motorcycles)
```

The code at ❶ tells Python to figure out where 'ducati' appears in the list and remove that element:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

You can also use the `remove()` method to work with a value that's being removed from a list. Let's remove the value 'ducati' and print a reason for removing it from the list:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

❷ too_expensive = 'ducati'
❸ motorcycles.remove(too_expensive)
print(motorcycles)
❹ print(f"\nA {too_expensive.title()} is too expensive for me.")
```

After defining the list at ❶, we assign the value 'ducati' to a variable called `too_expensive` ❷. We then use this variable to tell Python which value to remove from the list at ❸. At ❹ the value 'ducati' has been removed

from the list but is still accessible through the variable `too_expensive`, allowing us to print a statement about why we removed 'ducati' from the list of motorcycles:

```
['honda', 'yamaha', 'suzuki', 'ducati']  
['honda', 'yamaha', 'suzuki']
```

A Ducati is too expensive for me.

NOTE

The `remove()` method deletes only the first occurrence of the value you specify. If there's a possibility the value appears more than once in the list, you'll need to use a loop to make sure all occurrences of the value are removed. You'll learn how to do this in Chapter 7.

TRY IT YOURSELF

The following exercises are a bit more complex than those in Chapter 2, but they give you an opportunity to use lists in all of the ways described.

3-4. Guest List: If you could invite anyone, living or deceased, to dinner, who would you invite? Make a list that includes at least three people you'd like to invite to dinner. Then use your list to print a message to each person, inviting them to dinner.

3-5. Changing Guest List: You just heard that one of your guests can't make the dinner, so you need to send out a new set of invitations. You'll have to think of someone else to invite.

- Start with your program from Exercise 3-4. Add a `print()` call at the end of your program stating the name of the guest who can't make it.
- Modify your list, replacing the name of the guest who can't make it with the name of the new person you are inviting.
- Print a second set of invitation messages, one for each person who is still in your list.

3-6. More Guests: You just found a bigger dinner table, so now more space is available. Think of three more guests to invite to dinner.

- Start with your program from Exercise 3-4 or Exercise 3-5. Add a `print()` call to the end of your program informing people that you found a bigger dinner table.
- Use `insert()` to add one new guest to the beginning of your list.
- Use `insert()` to add one new guest to the middle of your list.
- Use `append()` to add one new guest to the end of your list.
- Print a new set of invitation messages, one for each person in your list.

3-7. Shrinking Guest List: You just found out that your new dinner table won't arrive in time for the dinner, and you have space for only two guests.

- Start with your program from Exercise 3-6. Add a new line that prints a message saying that you can invite only two people for dinner.
- Use `pop()` to remove guests from your list one at a time until only two names remain in your list. Each time you pop a name from your list, print a message to that person letting them know you're sorry you can't invite them to dinner.
- Print a message to each of the two people still on your list, letting them know they're still invited.
- Use `del` to remove the last two names from your list, so you have an empty list. Print your list to make sure you actually have an empty list at the end of your program.

Organizing a List

Often, your lists will be created in an unpredictable order, because you can't always control the order in which your users provide their data. Although this is unavoidable in most circumstances, you'll frequently want to present your information in a particular order. Sometimes you'll want to preserve the original order of your list, and other times you'll want to change the original order. Python provides a number of different ways to organize your lists, depending on the situation.

Sorting a List Permanently with the `sort()` Method

Python's `sort()` method makes it relatively easy to sort a list. Imagine we have a list of cars and want to change the order of the list to store them alphabetically. To keep the task simple, let's assume that all the values in the list are lowercase.

```
cars.py    cars = ['bmw', 'audi', 'toyota', 'subaru']
❶      cars.sort()
          print(cars)
```

The `sort()` method, shown at ❶, changes the order of the list permanently. The cars are now in alphabetical order, and we can never revert to the original order:

```
['audi', 'bmw', 'subaru', 'toyota']
```

You can also sort this list in reverse alphabetical order by passing the argument `reverse=True` to the `sort()` method. The following example sorts the list of cars in reverse alphabetical order:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort(reverse=True)
print(cars)
```

Again, the order of the list is permanently changed:

```
['toyota', 'subaru', 'bmw', 'audi']
```

Sorting a List Temporarily with the `sorted()` Function

To maintain the original order of a list but present it in a sorted order, you can use the `sorted()` function. The `sorted()` function lets you display your list in a particular order but doesn't affect the actual order of the list.

Let's try this function on the list of cars.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
```

- ❶ `print("Here is the original list:")`
`print(cars)`
 - ❷ `print("\nHere is the sorted list:")`
`print(sorted(cars))`
 - ❸ `print("\nHere is the original list again:")`
`print(cars)`
-

We first print the list in its original order at ❶ and then in alphabetical order at ❷. After the list is displayed in the new order, we show that the list is still stored in its original order at ❸.

Here is the original list:
['bmw', 'audi', 'toyota', 'subaru']

Here is the sorted list:
['audi', 'bmw', 'subaru', 'toyota']

- ❹ Here is the original list again:
['bmw', 'audi', 'toyota', 'subaru']
-

Notice that the list still exists in its original order at ❹ after the `sorted()` function has been used. The `sorted()` function can also accept a `reverse=True` argument if you want to display a list in reverse alphabetical order.

NOTE

Sorting a list alphabetically is a bit more complicated when all the values are not in lowercase. There are several ways to interpret capital letters when determining a sort order, and specifying the exact order can be more complex than we want to deal with at this time. However, most approaches to sorting will build directly on what you learned in this section.

Printing a List in Reverse Order

To reverse the original order of a list, you can use the `reverse()` method. If we originally stored the list of cars in chronological order according to when we owned them, we could easily rearrange the list into reverse chronological order:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)

cars.reverse()
print(cars)
```

Notice that `reverse()` doesn't sort backward alphabetically; it simply reverses the order of the list:

```
['bmw', 'audi', 'toyota', 'subaru']
['subaru', 'toyota', 'audi', 'bmw']
```

The `reverse()` method changes the order of a list permanently, but you can revert to the original order anytime by applying `reverse()` to the same list a second time.

Finding the Length of a List

You can quickly find the length of a list by using the `len()` function. The list in this example has four items, so its length is 4:

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(cars)
4
```

You'll find `len()` useful when you need to identify the number of aliens that still need to be shot down in a game, determine the amount of data you have to manage in a visualization, or figure out the number of registered users on a website, among other tasks.

NOTE

Python counts the items in a list starting with one, so you shouldn't run into any off-by-one errors when determining the length of a list.

TRY IT YOURSELF

3-8. Seeing the World: Think of at least five places in the world you'd like to visit.

- Store the locations in a list. Make sure the list is not in alphabetical order.
- Print your list in its original order. Don't worry about printing the list neatly, just print it as a raw Python list.
- Use `sorted()` to print your list in alphabetical order without modifying the actual list.
- Show that your list is still in its original order by printing it.
- Use `sorted()` to print your list in reverse alphabetical order without changing the order of the original list.
- Show that your list is still in its original order by printing it again.
- Use `reverse()` to change the order of your list. Print the list to show that its order has changed.
- Use `reverse()` to change the order of your list again. Print the list to show it's back to its original order.
- Use `sort()` to change your list so it's stored in alphabetical order. Print the list to show that its order has been changed.
- Use `sort()` to change your list so it's stored in reverse alphabetical order. Print the list to show that its order has changed.

3-9. Dinner Guests: Working with one of the programs from Exercises 3-4 through 3-7 (page 42), use `len()` to print a message indicating the number of people you are inviting to dinner.

3-10. Every Function: Think of something you could store in a list. For example, you could make a list of mountains, rivers, countries, cities, languages, or anything else you'd like. Write a program that creates a list containing these items and then uses each function introduced in this chapter at least once.

Avoiding Index Errors When Working with Lists

One type of error is common to see when you're working with lists for the first time. Let's say you have a list with three items, and you ask for the fourth item:

motorcycles.py

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[3])
```

This example results in an *index error*:

```
Traceback (most recent call last):
  File "motorcycles.py", line 2, in <module>
    print(motorcycles[3])
IndexError: list index out of range
```

Python attempts to give you the item at index 3. But when it searches the list, no item in `motorcycles` has an index of 3. Because of the off-by-one nature of indexing in lists, this error is typical. People think the third item is item number 3, because they start counting at 1. But in Python the third item is number 2, because it starts indexing at 0.

An index error means Python can't find an item at the index you requested. If an index error occurs in your program, try adjusting the index you're asking for by one. Then run the program again to see if the results are correct.

Keep in mind that whenever you want to access the last item in a list you use the index `-1`. This will always work, even if your list has changed size since the last time you accessed it:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[-1])
```

The index `-1` always returns the last item in a list, in this case the value `'suzuki'`:

```
'suzuki'
```

The only time this approach will cause an error is when you request the last item from an empty list:

```
motorcycles = []
print(motorcycles[-1])
```

No items are in `motorcycles`, so Python returns another index error:

```
Traceback (most recent call last):
  File "motorcycles.py", line 3, in <module>
    print(motorcycles[-1])
IndexError: list index out of range
```

NOTE

If an index error occurs and you can't figure out how to resolve it, try printing your list or just printing the length of your list. Your list might look much different than you thought it did, especially if it has been managed dynamically by your program. Seeing the actual list, or the exact number of items in your list, can help you sort out such logical errors.

TRY IT YOURSELF

3-11. Intentional Error: If you haven't received an index error in one of your programs yet, try to make one happen. Change an index in one of your programs to produce an index error. Make sure you correct the error before closing the program.

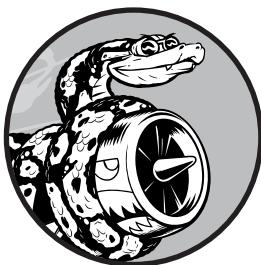
Summary

In this chapter you learned what lists are and how to work with the individual items in a list. You learned how to define a list and how to add and remove elements. You learned to sort lists permanently and temporarily for display purposes. You also learned how to find the length of a list and how to avoid index errors when you're working with lists.

In Chapter 4 you'll learn how to work with items in a list more efficiently. By looping through each item in a list using just a few lines of code you'll be able to work efficiently, even when your list contains thousands or millions of items.

4

WORKING WITH LISTS



In Chapter 3 you learned how to make a simple list, and you learned to work with the individual elements in a list. In this chapter you'll learn how to *loop* through an entire list using just a few lines of code regardless of how long the list is. Looping allows you to take the same action, or set of actions, with every item in a list. As a result, you'll be able to work efficiently with lists of any length, including those with thousands or even millions of items.

Looping Through an Entire List

You'll often want to run through all entries in a list, performing the same task with each item. For example, in a game you might want to move every element on the screen by the same amount, or in a list of numbers you might want to perform the same statistical operation on every element. Or perhaps you'll want to display each headline from a list of articles on a website. When you want to do the same action with every item in a list, you can use Python's `for` loop.

Let's say we have a list of magicians' names, and we want to print out each name in the list. We could do this by retrieving each name from the list individually, but this approach could cause several problems. For one, it would be repetitive to do this with a long list of names. Also, we'd have to change our code each time the list's length changed. A `for` loop avoids both of these issues by letting Python manage these issues internally.

Let's use a `for` loop to print out each name in a list of magicians:

```
magicians.py ❶ magicians = ['alice', 'david', 'carolina']
❷ for magician in magicians:
❸     print(magician)
```

We begin by defining a list at ❶, just as we did in Chapter 3. At ❷, we define a `for` loop. This line tells Python to pull a name from the list `magicians`, and associate it with the variable `magician`. At ❸ we tell Python to print the name that's just been assigned to `magician`. Python then repeats lines ❷ and ❸, once for each name in the list. It might help to read this code as "For every magician in the list of magicians, print the magician's name." The output is a simple printout of each name in the list:

```
alice
david
carolina
```

A Closer Look at Looping

The concept of looping is important because it's one of the most common ways a computer automates repetitive tasks. For example, in a simple loop like we used in `magicians.py`, Python initially reads the first line of the loop:

```
for magician in magicians:
```

This line tells Python to retrieve the first value from the list `magicians` and associate it with the variable `magician`. This first value is '`alice`'. Python then reads the next line:

```
    print(magician)
```

Python prints the current value of `magician`, which is still '`alice`'. Because the list contains more values, Python returns to the first line of the loop:

```
for magician in magicians:
```

Python retrieves the next name in the list, '`david`', and associates that value with the variable `magician`. Python then executes the line:

```
    print(magician)
```

Python prints the current value of `magician` again, which is now 'david'. Python repeats the entire loop once more with the last value in the list, 'carolina'. Because no more values are in the list, Python moves on to the next line in the program. In this case nothing comes after the `for` loop, so the program simply ends.

When you're using loops for the first time, keep in mind that the set of steps is repeated once for each item in the list, no matter how many items are in the list. If you have a million items in your list, Python repeats these steps a million times—and usually very quickly.

Also keep in mind when writing your own `for` loops that you can choose any name you want for the temporary variable that will be associated with each value in the list. However, it's helpful to choose a meaningful name that represents a single item from the list. For example, here's a good way to start a `for` loop for a list of cats, a list of dogs, and a general list of items:

```
for cat in cats:  
for dog in dogs:  
for item in list_of_items:
```

These naming conventions can help you follow the action being done on each item within a `for` loop. Using singular and plural names can help you identify whether a section of code is working with a single element from the list or the entire list.

Doing More Work Within a for Loop

You can do just about anything with each item in a `for` loop. Let's build on the previous example by printing a message to each magician, telling them that they performed a great trick:

magicians.py

```
magicians = ['alice', 'david', 'carolina']  
for magician in magicians:  
❶    print(f'{magician.title()}, that was a great trick!')
```

The only difference in this code is at ❶ where we compose a message to each magician, starting with that magician's name. The first time through the loop the value of `magician` is 'alice', so Python starts the first message with the name 'Alice'. The second time through the message will begin with 'David', and the third time through the message will begin with 'Carolina'.

The output shows a personalized message for each magician in the list:

```
Alice, that was a great trick!  
David, that was a great trick!  
Carolina, that was a great trick!
```

You can also write as many lines of code as you like in the `for` loop. Every indented line following the line `for magician in magicians` is considered *inside the loop*, and each indented line is executed once for each

value in the list. Therefore, you can do as much work as you like with each value in the list.

Let's add a second line to our message, telling each magician that we're looking forward to their next trick:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
❶    print(f"I can't wait to see your next trick, {magician.title()}.\\n")
```

Because we have indented both calls to `print()`, each line will be executed once for every magician in the list. The newline ("\\n") in the second `print()` call ❶ inserts a blank line after each pass through the loop. This creates a set of messages that are neatly grouped for each person in the list:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.
```

```
David, that was a great trick!
I can't wait to see your next trick, David.
```

```
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

You can use as many lines as you like in your `for` loops. In practice you'll often find it useful to do a number of different operations with each item in a list when you use a `for` loop.

Doing Something After a for Loop

What happens once a `for` loop has finished executing? Usually, you'll want to summarize a block of output or move on to other work that your program must accomplish.

Any lines of code after the `for` loop that are not indented are executed once without repetition. Let's write a thank you to the group of magicians as a whole, thanking them for putting on an excellent show. To display this group message after all of the individual messages have been printed, we place the thank you message after the `for` loop without indentation:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\\n")
❶ print("Thank you, everyone. That was a great magic show!")
```

The first two calls to `print()` are repeated once for each magician in the list, as you saw earlier. However, because the line at ❶ is not indented, it's printed only once:

Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you, everyone. That was a great magic show!

When you're processing data using a `for` loop, you'll find that this is a good way to summarize an operation that was performed on an entire data set. For example, you might use a `for` loop to initialize a game by running through a list of characters and displaying each character on the screen. You might then write some additional code after this loop that displays a *Play Now* button after all the characters have been drawn to the screen.

Avoiding Indentation Errors

Python uses indentation to determine how a line, or group of lines, is related to the rest of the program. In the previous examples, the lines that printed messages to individual magicians were part of the `for` loop because they were indented. Python's use of indentation makes code very easy to read. Basically, it uses whitespace to force you to write neatly formatted code with a clear visual structure. In longer Python programs, you'll notice blocks of code indented at a few different levels. These indentation levels help you gain a general sense of the overall program's organization.

As you begin to write code that relies on proper indentation, you'll need to watch for a few common *indentation errors*. For example, people sometimes indent lines of code that don't need to be indented or forget to indent lines that need to be indented. Seeing examples of these errors now will help you avoid them in the future and correct them when they do appear in your own programs.

Let's examine some of the more common indentation errors.

Forgetting to Indent

Always indent the line after the `for` statement in a loop. If you forget, Python will remind you:

```
magicians.py      magicians = ['alice', 'david', 'carolina']
                  for magician in magicians:
❶                   print(magician)
```

The call to `print()` ❶ should be indented, but it's not. When Python expects an indented block and doesn't find one, it lets you know which line it had a problem with.

```
File "magicians.py", line 3
    print(magician)
          ^
IndentationError: expected an indented block
```

You can usually resolve this kind of indentation error by indenting the line or lines immediately after the `for` statement.

Forgetting to Indent Additional Lines

Sometimes your loop will run without any errors but won't produce the expected result. This can happen when you're trying to do several tasks in a loop and you forget to indent some of its lines.

For example, this is what happens when we forget to indent the second line in the loop that tells each magician we're looking forward to their next trick:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
❶ print(f"I can't wait to see your next trick, {magician.title()}.\\n")
```

The call to `print()` ❶ is supposed to be indented, but because Python finds at least one indented line after the `for` statement, it doesn't report an error. As a result, the first `print()` call is executed once for each name in the list because it is indented. The second `print()` call is not indented, so it is executed only once after the loop has finished running. Because the final value associated with `magician` is '`carolina`', she is the only one who receives the "looking forward to the next trick" message:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

This is a *logical error*. The syntax is valid Python code, but the code does not produce the desired result because a problem occurs in its logic. If you expect to see a certain action repeated once for each item in a list and it's executed only once, determine whether you need to simply indent a line or a group of lines.

Indenting Unnecessarily

If you accidentally indent a line that doesn't need to be indented, Python informs you about the unexpected indent:

hello_world.py

```
message = "Hello Python world!"  
①     print(message)
```

We don't need to indent the `print()` call ①, because it isn't part of a loop; hence, Python reports that error:

```
File "hello_world.py", line 2  
    print(message)  
    ^  
IndentationError: unexpected indent
```

You can avoid unexpected indentation errors by indenting only when you have a specific reason to do so. In the programs you're writing at this point, the only lines you should indent are the actions you want to repeat for each item in a `for` loop.

Indenting Unnecessarily After the Loop

If you accidentally indent code that should run after a loop has finished, that code will be repeated once for each item in the list. Sometimes this prompts Python to report an error, but often this will result in a logical error.

For example, let's see what happens when we accidentally indent the line that thanked the magicians as a group for putting on a good show:

magicians.py

```
magicians = ['alice', 'david', 'carolina']  
for magician in magicians:  
    print(f"{magician.title()}, that was a great trick!")  
    print(f"I can't wait to see your next trick, {magician.title()}.\\n")  
①    print("Thank you everyone, that was a great magic show!")
```

Because the line at ① is indented, it's printed once for each person in the list, as shown here:

```
Alice, that was a great trick!  
I can't wait to see your next trick, Alice.
```

```
Thank you everyone, that was a great magic show!  
David, that was a great trick!  
I can't wait to see your next trick, David.
```

```
Thank you everyone, that was a great magic show!  
Carolina, that was a great trick!  
I can't wait to see your next trick, Carolina.
```

```
Thank you everyone, that was a great magic show!
```

This is another logical error, similar to the one in “Forgetting to Indent Additional Lines” on page 54. Because Python doesn’t know what you’re trying to accomplish with your code, it will run all code that is written in valid syntax. If an action is repeated many times when it should be executed only once, you probably need to unindent the code for that action.

Forgetting the Colon

The colon at the end of a `for` statement tells Python to interpret the next line as the start of a loop.

```
magicians = ['alice', 'david', 'carolina']
❶ for magician in magicians
    print(magician)
```

If you accidentally forget the colon, as shown at ❶, you’ll get a syntax error because Python doesn’t know what you’re trying to do. Although this is an easy error to fix, it’s not always an easy error to find. You’d be surprised by the amount of time programmers spend hunting down single-character errors like this. Such errors are difficult to find because we often just see what we expect to see.

TRY IT YOURSELF

4-1. Pizzas: Think of at least three kinds of your favorite pizza. Store these pizza names in a list, and then use a `for` loop to print the name of each pizza.

- Modify your `for` loop to print a sentence using the name of the pizza instead of printing just the name of the pizza. For each pizza you should have one line of output containing a simple statement like *I like pepperoni pizza*.
- Add a line at the end of your program, outside the `for` loop, that states how much you like pizza. The output should consist of three or more lines about the kinds of pizza you like and then an additional sentence, such as *I really love pizza!*

4-2. Animals: Think of at least three different animals that have a common characteristic. Store the names of these animals in a list, and then use a `for` loop to print out the name of each animal.

- Modify your program to print a statement about each animal, such as *A dog would make a great pet.*
- Add a line at the end of your program stating what these animals have in common. You could print a sentence such as *Any of these animals would make a great pet!*

Making Numerical Lists

Many reasons exist to store a set of numbers. For example, you'll need to keep track of the positions of each character in a game, and you might want to keep track of a player's high scores as well. In data visualizations, you'll almost always work with sets of numbers, such as temperatures, distances, population sizes, or latitude and longitude values, among other types of numerical sets.

Lists are ideal for storing sets of numbers, and Python provides a variety of tools to help you work efficiently with lists of numbers. Once you understand how to use these tools effectively, your code will work well even when your lists contain millions of items.

Using the range() Function

Python's `range()` function makes it easy to generate a series of numbers. For example, you can use the `range()` function to print a series of numbers like this:

```
first  
_numbers.py
```

```
for value in range(1, 5):  
    print(value)
```

Although this code looks like it should print the numbers from 1 to 5, it doesn't print the number 5:

```
1  
2  
3  
4
```

In this example, `range()` prints only the numbers 1 through 4. This is another result of the off-by-one behavior you'll see often in programming languages. The `range()` function causes Python to start counting at the first value you give it, and it stops when it reaches the second value you provide. Because it stops at that second value, the output never contains the end value, which would have been 5 in this case.

To print the numbers from 1 to 5, you would use `range(1, 6)`:

```
for value in range(1, 6):  
    print(value)
```

This time the output starts at 1 and ends at 5:

```
1  
2  
3  
4  
5
```

If your output is different than what you expect when you’re using `range()`, try adjusting your end value by 1.

You can also pass `range()` only one argument, and it will start the sequence of numbers at 0. For example, `range(6)` would return the numbers from 0 through 5.

Using `range()` to Make a List of Numbers

If you want to make a list of numbers, you can convert the results of `range()` directly into a list using the `list()` function. When you wrap `list()` around a call to the `range()` function, the output will be a list of numbers.

In the example in the previous section, we simply printed out a series of numbers. We can use `list()` to convert that same set of numbers into a list:

```
numbers = list(range(1, 6))
print(numbers)
```

And this is the result:

```
[1, 2, 3, 4, 5]
```

We can also use the `range()` function to tell Python to skip numbers in a given range. If you pass a third argument to `range()`, Python uses that value as a step size when generating numbers.

For example, here’s how to list the even numbers between 1 and 10:

```
even_numbers.py even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

In this example, the `range()` function starts with the value 2 and then adds 2 to that value. It adds 2 repeatedly until it reaches or passes the end value, 11, and produces this result:

```
[2, 4, 6, 8, 10]
```

You can create almost any set of numbers you want to using the `range()` function. For example, consider how you might make a list of the first 10 square numbers (that is, the square of each integer from 1 through 10). In Python, two asterisks (`**`) represent exponents. Here’s how you might put the first 10 square numbers into a list:

```
squares.py ❶ squares = []
❷ for value in range(1, 11):
❸     square = value ** 2
❹     squares.append(square)

❺ print(squares)
```

We start with an empty list called `squares` ❶. At ❷, we tell Python to loop through each value from 1 to 10 using the `range()` function. Inside the loop,

the current value is raised to the second power and assigned to the variable `square` ❸. At ❹, each new value of `square` is appended to the list `squares`. Finally, when the loop has finished running, the list of squares is printed ❺:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

To write this code more concisely, omit the temporary variable `square` and append each new value directly to the list:

```
squares = []
for value in range(1,11):
❶    squares.append(value**2)

print(squares)
```

The code at ❶ does the same work as the lines at ❸ and ❹ in `squares.py`. Each value in the loop is raised to the second power and then immediately appended to the list of squares.

You can use either of these two approaches when you’re making more complex lists. Sometimes using a temporary variable makes your code easier to read; other times it makes the code unnecessarily long. Focus first on writing code that you understand clearly, which does what you want it to do. Then look for more efficient approaches as you review your code.

Simple Statistics with a List of Numbers

A few Python functions are helpful when working with lists of numbers. For example, you can easily find the minimum, maximum, and sum of a list of numbers:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

NOTE

The examples in this section use short lists of numbers in order to fit easily on the page. They would work just as well if your list contained a million or more numbers.

List Comprehensions

The approach described earlier for generating the list `squares` consisted of using three or four lines of code. A *list comprehension* allows you to generate this same list in just one line of code. A list comprehension combines the `for` loop and the creation of new elements into one line, and automatically appends each new element. List comprehensions are not always presented to beginners, but I have included them here because you’ll most likely see them as soon as you start looking at other people’s code.

The following example builds the same list of square numbers you saw earlier but uses a list comprehension:

squares.py

```
squares = [value**2 for value in range(1, 11)]  
print(squares)
```

To use this syntax, begin with a descriptive name for the list, such as `squares`. Next, open a set of square brackets and define the expression for the values you want to store in the new list. In this example the expression is `value**2`, which raises the value to the second power. Then, write a `for` loop to generate the numbers you want to feed into the expression, and close the square brackets. The `for` loop in this example is `for value in range(1, 11)`, which feeds the values 1 through 10 into the expression `value**2`. Notice that no colon is used at the end of the `for` statement.

The result is the same list of square numbers you saw earlier:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

It takes practice to write your own list comprehensions, but you'll find them worthwhile once you become comfortable creating ordinary lists. When you're writing three or four lines of code to generate lists and it begins to feel repetitive, consider writing your own list comprehensions.

TRY IT YOURSELF

4-3. Counting to Twenty: Use a `for` loop to print the numbers from 1 to 20, inclusive.

4-4. One Million: Make a list of the numbers from one to one million, and then use a `for` loop to print the numbers. (If the output is taking too long, stop it by pressing `CTRL-C` or by closing the output window.)

4-5. Summing a Million: Make a list of the numbers from one to one million, and then use `min()` and `max()` to make sure your list actually starts at one and ends at one million. Also, use the `sum()` function to see how quickly Python can add a million numbers.

4-6. Odd Numbers: Use the third argument of the `range()` function to make a list of the odd numbers from 1 to 20. Use a `for` loop to print each number.

4-7. Threes: Make a list of the multiples of 3 from 3 to 30. Use a `for` loop to print the numbers in your list.

4-8. Cubes: A number raised to the third power is called a *cube*. For example, the cube of 2 is written as `2**3` in Python. Make a list of the first 10 cubes (that is, the cube of each integer from 1 through 10), and use a `for` loop to print out the value of each cube.

4-9. Cube Comprehension: Use a list comprehension to generate a list of the first 10 cubes.

Working with Part of a List

In Chapter 3 you learned how to access single elements in a list, and in this chapter you've been learning how to work through all the elements in a list. You can also work with a specific group of items in a list, which Python calls a *slice*.

Slicing a List

To make a slice, you specify the index of the first and last elements you want to work with. As with the `range()` function, Python stops one item before the second index you specify. To output the first three elements in a list, you would request indices 0 through 3, which would return elements 0, 1, and 2.

The following example involves a list of players on a team:

players.py

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
❶ print(players[0:3])
```

The code at ❶ prints a slice of this list, which includes just the first three players. The output retains the structure of the list and includes the first three players in the list:

```
['charles', 'martina', 'michael']
```

You can generate any subset of a list. For example, if you want the second, third, and fourth items in a list, you would start the slice at index 1 and end at index 4:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

This time the slice starts with 'martina' and ends with 'florence':

```
['martina', 'michael', 'florence']
```

If you omit the first index in a slice, Python automatically starts your slice at the beginning of the list:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[:4])
```

Without a starting index, Python starts at the beginning of the list:

```
['charles', 'martina', 'michael', 'florence']
```

A similar syntax works if you want a slice that includes the end of a list. For example, if you want all items from the third item through the last item, you can start with index 2 and omit the second index:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[2:])
```

Python returns all items from the third item through the end of the list:

```
['michael', 'florence', 'eli']
```

This syntax allows you to output all of the elements from any point in your list to the end regardless of the length of the list. Recall that a negative index returns an element a certain distance from the end of a list; therefore, you can output any slice from the end of a list. For example, if we want to output the last three players on the roster, we can use the slice `players[-3:]`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[-3:])
```

This prints the names of the last three players and would continue to work as the list of players changes in size.

NOTE

You can include a third value in the brackets indicating a slice. If a third value is included, this tells Python how many items to skip between items in the specified range.

Looping Through a Slice

You can use a slice in a `for` loop if you want to loop through a subset of the elements in a list. In the next example we loop through the first three players and print their names as part of a simple roster:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print("Here are the first three players on my team:")
❶ for player in players[:3]:
    print(player.title())
```

Instead of looping through the entire list of players at ❶, Python loops through only the first three names:

```
Here are the first three players on my team:
Charles
Martina
Michael
```

Slices are very useful in a number of situations. For instance, when you're creating a game, you could add a player's final score to a list every time that

player finishes playing. You could then get a player's top three scores by sorting the list in decreasing order and taking a slice that includes just the first three scores. When you're working with data, you can use slices to process your data in chunks of a specific size. Or, when you're building a web application, you could use slices to display information in a series of pages with an appropriate amount of information on each page.

Copying a List

Often, you'll want to start with an existing list and make an entirely new list based on the first one. Let's explore how copying a list works and examine one situation in which copying a list is useful.

To copy a list, you can make a slice that includes the entire original list by omitting the first index and the second index (`[:]`). This tells Python to make a slice that starts at the first item and ends with the last item, producing a copy of the entire list.

For example, imagine we have a list of our favorite foods and want to make a separate list of foods that a friend likes. This friend likes everything in our list so far, so we can create their list by copying ours:

```
foods.py ❶ my_foods = ['pizza', 'falafel', 'carrot cake']
❷ friend_foods = my_foods[:]

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

At ❶ we make a list of the foods we like called `my_foods`. At ❷ we make a new list called `friend_foods`. We make a copy of `my_foods` by asking for a slice of `my_foods` without specifying any indices and store the copy in `friend_foods`. When we print each list, we see that they both contain the same foods:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake']
```

```
My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake']
```

To prove that we actually have two separate lists, we'll add a new food to each list and show that each list keeps track of the appropriate person's favorite foods:

```
my_foods = ['pizza', 'falafel', 'carrot cake']
❶ friend_foods = my_foods[:]

❷ my_foods.append('cannoli')
❸ friend_foods.append('ice cream')
```

```
print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

At ❶ we copy the original items in `my_foods` to the new list `friend_foods`, as we did in the previous example. Next, we add a new food to each list: at ❷ we add 'cannoli' to `my_foods`, and at ❸ we add 'ice cream' to `friend_foods`. We then print the two lists to see whether each of these foods is in the appropriate list.

```
My favorite foods are:
❹ ['pizza', 'falafel', 'carrot cake', 'cannoli']
```

```
My friend's favorite foods are:
❺ ['pizza', 'falafel', 'carrot cake', 'ice cream']
```

The output at ❹ shows that 'cannoli' now appears in our list of favorite foods but 'ice cream' doesn't. At ❺ we can see that 'ice cream' now appears in our friend's list but 'cannoli' doesn't. If we had simply set `friend_foods` equal to `my_foods`, we would not produce two separate lists. For example, here's what happens when you try to copy a list without using a slice:

```
my_foods = ['pizza', 'falafel', 'carrot cake']

# This doesn't work:
❻ friend_foods = my_foods

my_foods.append('cannoli')
friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

Instead of storing a copy of `my_foods` in `friend_foods` at ❻, we set `friend_foods` equal to `my_foods`. This syntax actually tells Python to associate the new variable `friend_foods` with the list that is already associated with `my_foods`, so now both variables point to the same list. As a result, when we add 'cannoli' to `my_foods`, it will also appear in `friend_foods`. Likewise 'ice cream' will appear in both lists, even though it appears to be added only to `friend_foods`.

The output shows that both lists are the same now, which is not what we wanted:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']

NOTE

Don't worry about the details in this example for now. Basically, if you're trying to work with a copy of a list and you see unexpected behavior, make sure you are copying the list using a slice, as we did in the first example.

TRY IT YOURSELF

4-10. Slices: Using one of the programs you wrote in this chapter, add several lines to the end of the program that do the following:

- Print the message *The first three items in the list are:*. Then use a slice to print the first three items from that program's list.
- Print the message *Three items from the middle of the list are:*. Use a slice to print three items from the middle of the list.
- Print the message *The last three items in the list are:*. Use a slice to print the last three items in the list.

4-11. My Pizzas, Your Pizzas: Start with your program from Exercise 4-1 (page 56). Make a copy of the list of pizzas, and call it `friend_pizzas`. Then, do the following:

- Add a new pizza to the original list.
- Add a different pizza to the list `friend_pizzas`.
- Prove that you have two separate lists. Print the message *My favorite pizzas are:*, and then use a `for` loop to print the first list. Print the message *My friend's favorite pizzas are:*, and then use a `for` loop to print the second list. Make sure each new pizza is stored in the appropriate list.

4-12. More Loops: All versions of `foods.py` in this section have avoided using `for` loops when printing to save space. Choose a version of `foods.py`, and write two `for` loops to print each list of foods.

Tuples

Lists work well for storing collections of items that can change throughout the life of a program. The ability to modify lists is particularly important when you're working with a list of users on a website or a list of characters in a game. However, sometimes you'll want to create a list of items that cannot change. Tuples allow you to do just that. Python refers to values that cannot change as *immutable*, and an immutable list is called a *tuple*.

Defining a Tuple

A tuple looks just like a list except you use parentheses instead of square brackets. Once you define a tuple, you can access individual elements by using each item's index, just as you would for a list.

For example, if we have a rectangle that should always be a certain size, we can ensure that its size doesn't change by putting the dimensions into a tuple:

```
dimensions.py ❶ dimensions = (200, 50)
❷ print(dimensions[0])
print(dimensions[1])
```

We define the tuple `dimensions` at ❶, using parentheses instead of square brackets. At ❷ we print each element in the tuple individually, using the same syntax we've been using to access elements in a list:

```
200
50
```

Let's see what happens if we try to change one of the items in the tuple `dimensions`:

```
dimensions = (200, 50)
❶ dimensions[0] = 250
```

The code at ❶ tries to change the value of the first dimension, but Python returns a type error. Basically, because we're trying to alter a tuple, which can't be done to that type of object, Python tells us we can't assign a new value to an item in a tuple:

```
Traceback (most recent call last):
  File "dimensions.py", line 2, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

This is beneficial because we want Python to raise an error when a line of code tries to change the dimensions of the rectangle.

NOTE

Tuples are technically defined by the presence of a comma; the parentheses make them look neater and more readable. If you want to define a tuple with one element, you need to include a trailing comma:

```
my_t = (3,)
```

It doesn't often make sense to build a tuple with one element, but this can happen when tuples are generated automatically.

Looping Through All Values in a Tuple

You can loop over all the values in a tuple using a `for` loop, just as you did with a list:

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

Python returns all the elements in the tuple, just as it would for a list:

```
200
50
```

Writing over a Tuple

Although you can't modify a tuple, you can assign a new value to a variable that represents a tuple. So if we wanted to change our dimensions, we could redefine the entire tuple:

```
❶ dimensions = (200, 50)
print("Original dimensions:")
for dimension in dimensions:
    print(dimension)

❷ dimensions = (400, 100)
❸ print("\nModified dimensions:")
for dimension in dimensions:
    print(dimension)
```

The lines starting at ❶ define the original tuple and print the initial dimensions. At ❷, we associate a new tuple with the variable `dimensions`. We then print the new dimensions at ❸. Python doesn't raise any errors this time, because reassigning a variable is valid:

```
Original dimensions:
200
50
```

```
Modified dimensions:
400
100
```

When compared with lists, tuples are simple data structures. Use them when you want to store a set of values that should not be changed throughout the life of a program.

TRY IT YOURSELF

4-13. Buffet: A buffet-style restaurant offers only five basic foods. Think of five simple foods, and store them in a tuple.

- Use a `for` loop to print each food the restaurant offers.
- Try to modify one of the items, and make sure that Python rejects the change.
- The restaurant changes its menu, replacing two of the items with different foods. Add a line that rewrites the tuple, and then use a `for` loop to print each of the items on the revised menu.

Styling Your Code

Now that you’re writing longer programs, ideas about how to style your code are worthwhile to know. Take the time to make your code as easy as possible to read. Writing easy-to-read code helps you keep track of what your programs are doing and helps others understand your code as well.

Python programmers have agreed on a number of styling conventions to ensure that everyone’s code is structured in roughly the same way. Once you’ve learned to write clean Python code, you should be able to understand the overall structure of anyone else’s Python code, as long as they follow the same guidelines. If you’re hoping to become a professional programmer at some point, you should begin following these guidelines as soon as possible to develop good habits.

The Style Guide

When someone wants to make a change to the Python language, they write a *Python Enhancement Proposal (PEP)*. One of the oldest PEPs is *PEP 8*, which instructs Python programmers on how to style their code. *PEP 8* is fairly lengthy, but much of it relates to more complex coding structures than what you’ve seen so far.

The Python style guide was written with the understanding that code is read more often than it is written. You’ll write your code once and then start reading it as you begin debugging. When you add features to a program, you’ll spend more time reading your code. When you share your code with other programmers, they’ll read your code as well.

Given the choice between writing code that’s easier to write or code that’s easier to read, Python programmers will almost always encourage you to write code that’s easier to read. The following guidelines will help you write clear code from the start.

Indentation

PEP 8 recommends that you use four spaces per indentation level. Using four spaces improves readability while leaving room for multiple levels of indentation on each line.

In a word processing document, people often use tabs rather than spaces to indent. This works well for word processing documents, but the Python interpreter gets confused when tabs are mixed with spaces. Every text editor provides a setting that lets you use the TAB key but then converts each tab to a set number of spaces. You should definitely use your TAB key, but also make sure your editor is set to insert spaces rather than tabs into your document.

Mixing tabs and spaces in your file can cause problems that are very difficult to diagnose. If you think you have a mix of tabs and spaces, you can convert all tabs in a file to spaces in most editors.

Line Length

Many Python programmers recommend that each line should be less than 80 characters. Historically, this guideline developed because most computers could fit only 79 characters on a single line in a terminal window. Currently, people can fit much longer lines on their screens, but other reasons exist to adhere to the 79-character standard line length. Professional programmers often have several files open on the same screen, and using the standard line length allows them to see entire lines in two or three files that are open side by side onscreen. PEP 8 also recommends that you limit all of your comments to 72 characters per line, because some of the tools that generate automatic documentation for larger projects add formatting characters at the beginning of each commented line.

The PEP 8 guidelines for line length are not set in stone, and some teams prefer a 99-character limit. Don't worry too much about line length in your code as you're learning, but be aware that people who are working collaboratively almost always follow the PEP 8 guidelines. Most editors allow you to set up a visual cue, usually a vertical line on your screen, that shows you where these limits are.

NOTE

Appendix B shows you how to configure your text editor so it always inserts four spaces each time you press the TAB key and shows a vertical guideline to help you follow the 79-character limit.

Blank Lines

To group parts of your program visually, use blank lines. You should use blank lines to organize your files, but don't do so excessively. By following the examples provided in this book, you should strike the right balance. For example, if you have five lines of code that build a list, and then another three lines that do something with that list, it's appropriate to place a blank line between the two sections. However, you should not place three or four blank lines between the two sections.

Blank lines won't affect how your code runs, but they will affect the readability of your code. The Python interpreter uses horizontal indentation to interpret the meaning of your code, but it disregards vertical spacing.

Other Style Guidelines

PEP 8 has many additional styling recommendations, but most of the guidelines refer to more complex programs than what you're writing at this point. As you learn more complex Python structures, I'll share the relevant parts of the PEP 8 guidelines.

TRY IT YOURSELF

4-14. PEP 8: Look through the original PEP 8 style guide at <https://python.org/dev/peps/pep-0008/>. You won't use much of it now, but it might be interesting to skim through it.

4-15. Code Review: Choose three of the programs you've written in this chapter and modify each one to comply with PEP 8:

- Use four spaces for each indentation level. Set your text editor to insert four spaces every time you press TAB, if you haven't already done so (see Appendix B for instructions on how to do this).
- Use less than 80 characters on each line, and set your editor to show a vertical guideline at the 80th character position.
- Don't use blank lines excessively in your program files.

Summary

In this chapter you learned how to work efficiently with the elements in a list. You learned how to work through a list using a `for` loop, how Python uses indentation to structure a program, and how to avoid some common indentation errors. You learned to make simple numerical lists, as well as a few operations you can perform on numerical lists. You learned how to slice a list to work with a subset of items and how to copy lists properly using a slice. You also learned about tuples, which provide a degree of protection to a set of values that shouldn't change, and how to style your increasingly complex code to make it easy to read.

In Chapter 5, you'll learn to respond appropriately to different conditions by using `if` statements. You'll learn to string together relatively complex sets of conditional tests to respond appropriately to exactly the kind of situation or information you're looking for. You'll also learn to use `if` statements while looping through a list to take specific actions with selected elements from a list.

5

IF STATEMENTS



Programming often involves examining a set of conditions and deciding which action to take based on those conditions.

Python’s if statement allows you to examine the current state of a program and respond appropriately to that state.

In this chapter you’ll learn to write conditional tests, which allow you to check any condition of interest. You’ll learn to write simple if statements, and you’ll learn how to create a more complex series of if statements to identify when the exact conditions you want are present. You’ll then apply this concept to lists, so you’ll be able to write a for loop that handles most items in a list one way but handles certain items with specific values in a different way.

A Simple Example

The following short example shows how if tests let you respond to special situations correctly. Imagine you have a list of cars and you want to print out the name of each car. Car names are proper names, so the names of most cars should be printed in title case. However, the value 'bmw' should be printed in all uppercase. The following code loops through a list of car names and looks for the value 'bmw'. Whenever the value is 'bmw', it's printed in uppercase instead of title case:

```
cars.py    cars = ['audi', 'bmw', 'subaru', 'toyota']

        for car in cars:
❶            if car == 'bmw':
                print(car.upper())
            else:
                print(car.title())
```

The loop in this example first checks if the current value of car is 'bmw' **❶**. If it is, the value is printed in uppercase. If the value of car is anything other than 'bmw', it's printed in title case:

```
Audi
BMW
Subaru
Toyota
```

This example combines a number of the concepts you'll learn about in this chapter. Let's begin by looking at the kinds of tests you can use to examine the conditions in your program.

Conditional Tests

At the heart of every if statement is an expression that can be evaluated as True or False and is called a *conditional test*. Python uses the values True and False to decide whether the code in an if statement should be executed. If a conditional test evaluates to True, Python executes the code following the if statement. If the test evaluates to False, Python ignores the code following the if statement.

Checking for Equality

Most conditional tests compare the current value of a variable to a specific value of interest. The simplest conditional test checks whether the value of a variable is equal to the value of interest:

```
❶ >>> car = 'bmw'
❷ >>> car == 'bmw'
True
```

The line at ❶ sets the value of car to 'bmw' using a single equal sign, as you've seen many times already. The line at ❷ checks whether the value of car is 'bmw' using a double equal sign (==). This *equality operator* returns True if the values on the left and right side of the operator match, and False if they don't match. The values in this example match, so Python returns True.

When the value of car is anything other than 'bmw', this test returns False:

```
❶ >>> car = 'audi'  
❷ >>> car == 'bmw'  
False
```

A single equal sign is really a statement; you might read the code at ❶ as "Set the value of car equal to 'audi'." On the other hand, a double equal sign, like the one at ❷, asks a question: "Is the value of car equal to 'bmw'?" Most programming languages use equal signs in this way.

Ignoring Case When Checking for Equality

Testing for equality is case sensitive in Python. For example, two values with different capitalization are not considered equal:

```
>>> car = 'Audi'  
>>> car == 'audi'  
False
```

If case matters, this behavior is advantageous. But if case doesn't matter and instead you just want to test the value of a variable, you can convert the variable's value to lowercase before doing the comparison:

```
>>> car = 'Audi'  
>>> car.lower() == 'audi'  
True
```

This test would return True no matter how the value 'Audi' is formatted because the test is now case insensitive. The lower() function doesn't change the value that was originally stored in car, so you can do this kind of comparison without affecting the original variable:

```
❶ >>> car = 'Audi'  
❷ >>> car.lower() == 'audi'  
True  
❸ >>> car  
'Audi'
```

At ❶ we assign the capitalized string 'Audi' to the variable car. At ❷ we convert the value of car to lowercase and compare the lowercase value

to the string 'audi'. The two strings match, so Python returns `True`. At ❸ we can see that the value stored in `car` has not been affected by the `lower()` method.

Websites enforce certain rules for the data that users enter in a manner similar to this. For example, a site might use a conditional test like this to ensure that every user has a truly unique username, not just a variation on the capitalization of another person's username. When someone submits a new username, that new username is converted to lowercase and compared to the lowercase versions of all existing usernames. During this check, a username like 'John' will be rejected if any variation of 'john' is already in use.

Checking for Inequality

When you want to determine whether two values are not equal, you can combine an exclamation point and an equal sign (`!=`). The exclamation point represents *not*, as it does in many programming languages.

Let's use another `if` statement to examine how to use the inequality operator. We'll store a requested pizza topping in a variable and then print a message if the person did not order anchovies:

```
toppings.py     requested_topping = 'mushrooms'

❶ if requested_topping != 'anchovies':
    print("Hold the anchovies!")
```

The line at ❶ compares the value of `requested_topping` to the value '`anchovies`'. If these two values do not match, Python returns `True` and executes the code following the `if` statement. If the two values match, Python returns `False` and does not run the code following the `if` statement.

Because the value of `requested_topping` is not '`anchovies`', the `print()` function is executed:

```
Hold the anchovies!
```

Most of the conditional expressions you write will test for equality, but sometimes you'll find it more efficient to test for inequality.

Numerical Comparisons

Testing numerical values is pretty straightforward. For example, the following code checks whether a person is 18 years old:

```
>>> age = 18
>>> age == 18
True
```

You can also test to see if two numbers are not equal. For example, the following code prints a message if the given answer is not correct:

```
magic  
_number.py  
❶ if answer != 42:  
    print("That is not the correct answer. Please try again!")
```

The conditional test at ❶ passes, because the value of `answer` (17) is not equal to 42. Because the test passes, the indented code block is executed:

```
That is not the correct answer. Please try again!
```

You can include various mathematical comparisons in your conditional statements as well, such as less than, less than or equal to, greater than, and greater than or equal to:

```
>>> age = 19  
>>> age < 21  
True  
>>> age <= 21  
True  
>>> age > 21  
False  
>>> age >= 21  
False
```

Each mathematical comparison can be used as part of an `if` statement, which can help you detect the exact conditions of interest.

Checking Multiple Conditions

You may want to check multiple conditions at the same time. For example, sometimes you might need two conditions to be `True` to take an action. Other times you might be satisfied with just one condition being `True`. The keywords `and` and `or` can help you in these situations.

Using `and` to Check Multiple Conditions

To check whether two conditions are both `True` simultaneously, use the keyword `and` to combine the two conditional tests; if each test passes, the overall expression evaluates to `True`. If either test fails or if both tests fail, the expression evaluates to `False`.

For example, you can check whether two people are both over 21 using the following test:

```
❶ >>> age_0 = 22  
>>> age_1 = 18  
❷ >>> age_0 >= 21 and age_1 >= 21  
False
```

```
❸ >>> age_1 = 22
>>> age_0 >= 21 and age_1 >= 21
True
```

At ❶ we define two ages, `age_0` and `age_1`. At ❷ we check whether both ages are 21 or older. The test on the left passes, but the test on the right fails, so the overall conditional expression evaluates to `False`. At ❸ we change `age_1` to 22. The value of `age_1` is now greater than 21, so both individual tests pass, causing the overall conditional expression to evaluate as `True`.

To improve readability, you can use parentheses around the individual tests, but they are not required. If you use parentheses, your test would look like this:

```
(age_0 >= 21) and (age_1 >= 21)
```

Using `or` to Check Multiple Conditions

The keyword `or` allows you to check multiple conditions as well, but it passes when either or both of the individual tests pass. An `or` expression fails only when both individual tests fail.

Let's consider two ages again, but this time we'll look for only one person to be over 21:

```
❶ >>> age_0 = 22
>>> age_1 = 18
❷ >>> age_0 >= 21 or age_1 >= 21
True
❸ >>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

We start with two age variables again at ❶. Because the test for `age_0` at ❷ passes, the overall expression evaluates to `True`. We then lower `age_0` to 18. In the test at ❸, both tests now fail and the overall expression evaluates to `False`.

Checking Whether a Value Is in a List

Sometimes it's important to check whether a list contains a certain value before taking an action. For example, you might want to check whether a new username already exists in a list of current usernames before completing someone's registration on a website. In a mapping project, you might want to check whether a submitted location already exists in a list of known locations.

To find out whether a particular value is already in a list, use the keyword `in`. Let's consider some code you might write for a pizzeria. We'll make a list of toppings a customer has requested for a pizza and then check whether certain toppings are in the list.

```
>>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
❶ >>> 'mushrooms' in requested_toppings
True
❷ >>> 'pepperoni' in requested_toppings
False
```

At ❶ and ❷, the keyword `in` tells Python to check for the existence of '`mushrooms`' and '`pepperoni`' in the list `requested_toppings`. This technique is quite powerful because you can create a list of essential values, and then easily check whether the value you're testing matches one of the values in the list.

Checking Whether a Value Is Not in a List

Other times, it's important to know if a value does not appear in a list. You can use the keyword `not` in this situation. For example, consider a list of users who are banned from commenting in a forum. You can check whether a user has been banned before allowing that person to submit a comment:

```
banned
users.py
```

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'
```

```
❶ if user not in banned_users:
    print(f"{user.title()}, you can post a response if you wish.")
```

The line at ❶ reads quite clearly. If the value of `user` is not in the list `banned_users`, Python returns `True` and executes the indented line.

The user '`marie`' is not in the list `banned_users`, so she sees a message inviting her to post a response:

```
Marie, you can post a response if you wish.
```

Boolean Expressions

As you learn more about programming, you'll hear the term *Boolean expression* at some point. A Boolean expression is just another name for a conditional test. A *Boolean value* is either `True` or `False`, just like the value of a conditional expression after it has been evaluated.

Boolean values are often used to keep track of certain conditions, such as whether a game is running or whether a user can edit certain content on a website:

```
game_active = True
can_edit = False
```

Boolean values provide an efficient way to track the state of a program or a particular condition that is important in your program.

TRY IT YOURSELF

5-1. Conditional Tests: Write a series of conditional tests. Print a statement describing each test and your prediction for the results of each test. Your code should look something like this:

```
car = 'subaru'
print("Is car == 'subaru'? I predict True.")
print(car == 'subaru')

print("\nIs car == 'audi'? I predict False.")
print(car == 'audi')
```

- Look closely at your results, and make sure you understand why each line evaluates to True or False.
- Create at least ten tests. Have at least five tests evaluate to True and another five tests evaluate to False.

5-2. More Conditional Tests: You don't have to limit the number of tests you create to ten. If you want to try more comparisons, write more tests and add them to *conditional_tests.py*. Have at least one True and one False result for each of the following:

- Tests for equality and inequality with strings
- Tests using the `lower()` method
- Numerical tests involving equality and inequality, greater than and less than, greater than or equal to, and less than or equal to
- Tests using the `and` keyword and the `or` keyword
- Test whether an item is in a list
- Test whether an item is not in a list

if Statements

When you understand conditional tests, you can start writing if statements. Several different kinds of if statements exist, and your choice of which to use depends on the number of conditions you need to test. You saw several examples of if statements in the discussion about conditional tests, but now let's dig deeper into the topic.

Simple if Statements

The simplest kind of if statement has one test and one action:

```
if conditional_test:
    do something
```

You can put any conditional test in the first line and just about any action in the indented block following the test. If the conditional test evaluates to `True`, Python executes the code following the `if` statement. If the test evaluates to `False`, Python ignores the code following the `if` statement.

Let's say we have a variable representing a person's age, and we want to know if that person is old enough to vote. The following code tests whether the person can vote:

```
voting.py
age = 19
❶ if age >= 18:
❷     print("You are old enough to vote!")
```

At ❶ Python checks to see whether the value of `age` is greater than or equal to 18. It is, so Python executes the indented `print()` call at ❷:

```
You are old enough to vote!
```

Indentation plays the same role in `if` statements as it did in `for` loops. All indented lines after an `if` statement will be executed if the test passes, and the entire block of indented lines will be ignored if the test does not pass.

You can have as many lines of code as you want in the block following the `if` statement. Let's add another line of output if the person is old enough to vote, asking if the individual has registered to vote yet:

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

The conditional test passes, and both `print()` calls are indented, so both lines are printed:

```
You are old enough to vote!
Have you registered to vote yet?
```

If the value of `age` is less than 18, this program would produce no output.

if-else Statements

Often, you'll want to take one action when a conditional test passes and a different action in all other cases. Python's `if-else` syntax makes this possible. An `if-else` block is similar to a simple `if` statement, but the `else` statement allows you to define an action or set of actions that are executed when the conditional test fails.

We'll display the same message we had previously if the person is old enough to vote, but this time we'll add a message for anyone who is not old enough to vote:

```
age = 17
❶ if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
❷ else:
    print("Sorry, you are too young to vote.")
    print("Please register to vote as soon as you turn 18!")
```

If the conditional test at ❶ passes, the first block of indented `print()` calls is executed. If the test evaluates to `False`, the `else` block at ❷ is executed. Because `age` is less than 18 this time, the conditional test fails and the code in the `else` block is executed:

```
Sorry, you are too young to vote.
Please register to vote as soon as you turn 18!
```

This code works because it has only two possible situations to evaluate: a person is either old enough to vote or not old enough to vote. The `if-else` structure works well in situations in which you want Python to always execute one of two possible actions. In a simple `if-else` chain like this, one of the two actions will always be executed.

The if-elif-else Chain

Often, you'll need to test more than two possible situations, and to evaluate these you can use Python's `if-elif-else` syntax. Python executes only one block in an `if-elif-else` chain. It runs each conditional test in order until one passes. When a test passes, the code following that test is executed and Python skips the rest of the tests.

Many real-world situations involve more than two possible conditions. For example, consider an amusement park that charges different rates for different age groups:

- Admission for anyone under age 4 is free.
- Admission for anyone between the ages of 4 and 18 is \$25.
- Admission for anyone age 18 or older is \$40.

How can we use an `if` statement to determine a person's admission rate? The following code tests for the age group of a person and then prints an admission price message:

```
amusement
_park.py
age = 12
❶ if age < 4:
    print("Your admission cost is $0.")
```

```
❷ elif age < 18:  
    print("Your admission cost is $25.")  
❸ else:  
    print("Your admission cost is $40.")
```

The if test at ❶ tests whether a person is under 4 years old. If the test passes, an appropriate message is printed and Python skips the rest of the tests. The elif line at ❷ is really another if test, which runs only if the previous test failed. At this point in the chain, we know the person is at least 4 years old because the first test failed. If the person is under 18, an appropriate message is printed and Python skips the else block. If both the if and elif tests fail, Python runs the code in the else block at ❸.

In this example the test at ❶ evaluates to False, so its code block is not executed. However, the second test evaluates to True (12 is less than 18) so its code is executed. The output is one sentence, informing the user of the admission cost:

```
Your admission cost is $25.
```

Any age greater than 17 would cause the first two tests to fail. In these situations, the else block would be executed and the admission price would be \$40.

Rather than printing the admission price within the if-elif-else block, it would be more concise to set just the price inside the if-elif-else chain and then have a simple `print()` call that runs after the chain has been evaluated:

```
age = 12  
  
if age < 4:  
❶    price = 0  
elif age < 18:  
❷    price = 25  
else:  
❸    price = 40  
  
❹ print(f"Your admission cost is ${price}.")
```

The lines at ❶, ❷, and ❸ set the value of `price` according to the person's age, as in the previous example. After the price is set by the if-elif-else chain, a separate unindented `print()` call ❹ uses this value to display a message reporting the person's admission price.

This code produces the same output as the previous example, but the purpose of the if-elif-else chain is narrower. Instead of determining a price and displaying a message, it simply determines the admission price. In addition to being more efficient, this revised code is easier to modify than the original approach. To change the text of the output message, you would need to change only one `print()` call rather than three separate `print()` calls.

Using Multiple `elif` Blocks

You can use as many `elif` blocks in your code as you like. For example, if the amusement park were to implement a discount for seniors, you could add one more conditional test to the code to determine whether someone qualified for the senior discount. Let's say that anyone 65 or older pays half the regular admission, or \$20:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
❶ elif age < 65:
    price = 40
❷ else:
    price = 20

print(f"Your admission cost is ${price}.")
```

Most of this code is unchanged. The second `elif` block at ❶ now checks to make sure a person is less than age 65 before assigning them the full admission rate of \$40. Notice that the value assigned in the `else` block at ❷ needs to be changed to \$20, because the only ages that make it to this block are people 65 or older.

Omitting the `else` Block

Python does not require an `else` block at the end of an `if-elif` chain. Sometimes an `else` block is useful; sometimes it is clearer to use an additional `elif` statement that catches the specific condition of interest:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
❶ elif age >= 65:
    price = 20

print(f"Your admission cost is ${price}.")
```

The extra `elif` block at ❶ assigns a price of \$20 when the person is 65 or older, which is a bit clearer than the general `else` block. With this change, every block of code must pass a specific test in order to be executed.

The `else` block is a catchall statement. It matches any condition that wasn't matched by a specific `if` or `elif` test, and that can sometimes include invalid or even malicious data. If you have a specific final condition you are testing for, consider using a final `elif` block and omit the `else` block. As a result, you'll gain extra confidence that your code will run only under the correct conditions.

Testing Multiple Conditions

The `if-elif-else` chain is powerful, but it's only appropriate to use when you just need one test to pass. As soon as Python finds one test that passes, it skips the rest of the tests. This behavior is beneficial, because it's efficient and allows you to test for one specific condition.

However, sometimes it's important to check all of the conditions of interest. In this case, you should use a series of simple `if` statements with no `elif` or `else` blocks. This technique makes sense when more than one condition could be `True`, and you want to act on every condition that is `True`.

Let's reconsider the pizzeria example. If someone requests a two-topping pizza, you'll need to be sure to include both toppings on their pizza:

```
toppings.py ❶ requested_toppings = ['mushrooms', 'extra cheese']

❷ if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
❸ if 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
❹ if 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

We start at ❶ with a list containing the requested toppings. The `if` statement at ❷ checks to see whether the person requested mushrooms on their pizza. If so, a message is printed confirming that topping. The test for pepperoni at ❸ is another simple `if` statement, not an `elif` or `else` statement, so this test is run regardless of whether the previous test passed or not. The code at ❹ checks whether extra cheese was requested regardless of the results from the first two tests. These three independent tests are executed every time this program is run.

Because every condition in this example is evaluated, both mushrooms and extra cheese are added to the pizza:

```
Adding mushrooms.
Adding extra cheese.
```

```
Finished making your pizza!
```

This code would not work properly if we used an `if-elif-else` block, because the code would stop running after only one test passes. Here's what that would look like:

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

The test for '`mushrooms`' is the first test to pass, so mushrooms are added to the pizza. However, the values '`extra cheese`' and '`pepperoni`' are never checked, because Python doesn't run any tests beyond the first test that passes in an `if-elif-else` chain. The customer's first topping will be added, but all of their other toppings will be missed:

Adding mushrooms.

Finished making your pizza!

In summary, if you want only one block of code to run, use an `if-elif-else` chain. If more than one block of code needs to run, use a series of independent `if` statements.

TRY IT YOURSELF

5-3. Alien Colors #1: Imagine an alien was just shot down in a game. Create a variable called `alien_color` and assign it a value of '`green`', '`yellow`', or '`red`'.

- Write an `if` statement to test whether the alien's color is `green`. If it is, print a message that the player just earned 5 points.
- Write one version of this program that passes the `if` test and another that fails. (The version that fails will have no output.)

5-4. Alien Colors #2: Choose a color for an alien as you did in Exercise 5-3, and write an `if-else` chain.

- If the alien's color is `green`, print a statement that the player just earned 5 points for shooting the alien.
- If the alien's color isn't `green`, print a statement that the player just earned 10 points.
- Write one version of this program that runs the `if` block and another that runs the `else` block.

5-5. Alien Colors #3: Turn your if-else chain from Exercise 5-4 into an if-elif-else chain.

- If the alien is green, print a message that the player earned 5 points.
- If the alien is yellow, print a message that the player earned 10 points.
- If the alien is red, print a message that the player earned 15 points.
- Write three versions of this program, making sure each message is printed for the appropriate color alien.

5-6. Stages of Life: Write an if-elif-else chain that determines a person's stage of life. Set a value for the variable age, and then:

- If the person is less than 2 years old, print a message that the person is a baby.
- If the person is at least 2 years old but less than 4, print a message that the person is a toddler.
- If the person is at least 4 years old but less than 13, print a message that the person is a kid.
- If the person is at least 13 years old but less than 20, print a message that the person is a teenager.
- If the person is at least 20 years old but less than 65, print a message that the person is an adult.
- If the person is age 65 or older, print a message that the person is an elder.

5-7. Favorite Fruit: Make a list of your favorite fruits, and then write a series of independent if statements that check for certain fruits in your list.

- Make a list of your three favorite fruits and call it favorite_fruits.
- Write five if statements. Each should check whether a certain kind of fruit is in your list. If the fruit is in your list, the if block should print a statement, such as *You really like bananas!*

Using if Statements with Lists

You can do some interesting work when you combine lists and if statements. You can watch for special values that need to be treated differently than other values in the list. You can manage changing conditions efficiently, such as the availability of certain items in a restaurant throughout a shift. You can also begin to prove that your code works as you expect it to in all possible situations.

Checking for Special Items

This chapter began with a simple example that showed how to handle a special value like 'bmw', which needed to be printed in a different format than other values in the list. Now that you have a basic understanding of conditional tests and if statements, let's take a closer look at how you can watch for special values in a list and handle those values appropriately.

Let's continue with the pizzeria example. The pizzeria displays a message whenever a topping is added to your pizza, as it's being made. The code for this action can be written very efficiently by making a list of toppings the customer has requested and using a loop to announce each topping as it's added to the pizza:

toppings.py

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

The output is straightforward because this code is just a simple for loop:

```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.
```

Finished making your pizza!

But what if the pizzeria runs out of green peppers? An if statement inside the for loop can handle this situation appropriately:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    if requested_topping == 'green peppers':
        print("Sorry, we are out of green peppers right now.")
    else:
        print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

This time we check each requested item before adding it to the pizza. The code at ❶ checks to see if the person requested green peppers. If so, we display a message informing them why they can't have green peppers. The else block at ❷ ensures that all other toppings will be added to the pizza.

The output shows that each requested topping is handled appropriately.

```
Adding mushrooms.  
Sorry, we are out of green peppers right now.  
Adding extra cheese.
```

```
Finished making your pizza!
```

Checking That a List Is Not Empty

We've made a simple assumption about every list we've worked with so far; we've assumed that each list has at least one item in it. Soon we'll let users provide the information that's stored in a list, so we won't be able to assume that a list has any items in it each time a loop is run. In this situation, it's useful to check whether a list is empty before running a for loop.

As an example, let's check whether the list of requested toppings is empty before building the pizza. If the list is empty, we'll prompt the user and make sure they want a plain pizza. If the list is not empty, we'll build the pizza just as we did in the previous examples:

```
❶ requested_toppings = []  
  
❷ if requested_toppings:  
    for requested_topping in requested_toppings:  
        print(f"Adding {requested_topping}.")  
    print("\nFinished making your pizza!")  
❸ else:  
    print("Are you sure you want a plain pizza?")
```

This time we start out with an empty list of requested toppings at ❶. Instead of jumping right into a for loop, we do a quick check at ❷. When the name of a list is used in an if statement, Python returns True if the list contains at least one item; an empty list evaluates to False. If requested_toppings passes the conditional test, we run the same for loop we used in the previous example. If the conditional test fails, we print a message asking the customer if they really want a plain pizza with no toppings ❸.

The list is empty in this case, so the output asks if the user really wants a plain pizza:

```
Are you sure you want a plain pizza?
```

If the list is not empty, the output will show each requested topping being added to the pizza.

Using Multiple Lists

People will ask for just about anything, especially when it comes to pizza toppings. What if a customer actually wants french fries on their pizza? You can use lists and if statements to make sure your input makes sense before you act on it.

Let's watch out for unusual topping requests before we build a pizza. The following example defines two lists. The first is a list of available toppings at the pizzeria, and the second is the list of toppings that the user has requested. This time, each item in `requested_toppings` is checked against the list of available toppings before it's added to the pizza:

```
❶ available_toppings = ['mushrooms', 'olives', 'green peppers',
                        'pepperoni', 'pineapple', 'extra cheese']

❷ requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

❸ for requested_topping in requested_toppings:
❹     if requested_topping in available_toppings:
❺         print(f"Adding {requested_topping}.")
❻     else:
❼         print(f"Sorry, we don't have {requested_topping}.")

print("\nFinished making your pizza!")
```

At ❶ we define a list of available toppings at this pizzeria. Note that this could be a tuple if the pizzeria has a stable selection of toppings. At ❷, we make a list of toppings that a customer has requested. Note the unusual request, 'french fries'. At ❸ we loop through the list of requested toppings. Inside the loop, we first check to see if each requested topping is actually in the list of available toppings ❹. If it is, we add that topping to the pizza. If the requested topping is not in the list of available toppings, the else block will run ❽. The else block prints a message telling the user which toppings are unavailable.

This code syntax produces clean, informative output:

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

In just a few lines of code, we've managed a real-world situation pretty effectively!

TRY IT YOURSELF

5-8. Hello Admin: Make a list of five or more usernames, including the name 'admin'. Imagine you are writing code that will print a greeting to each user after they log in to a website. Loop through the list, and print a greeting to each user:

- If the username is 'admin', print a special greeting, such as *Hello admin, would you like to see a status report?*
- Otherwise, print a generic greeting, such as *Hello Jaden, thank you for logging in again.*

5-9. No Users: Add an if test to `hello_admin.py` to make sure the list of users is not empty.

- If the list is empty, print the message *We need to find some users!*
- Remove all of the usernames from your list, and make sure the correct message is printed.

5-10. Checking Usernames: Do the following to create a program that simulates how websites ensure that everyone has a unique username.

- Make a list of five or more usernames called `current_users`.
- Make another list of five usernames called `new_users`. Make sure one or two of the new usernames are also in the `current_users` list.
- Loop through the `new_users` list to see if each new username has already been used. If it has, print a message that the person will need to enter a new username. If a username has not been used, print a message saying that the username is available.
- Make sure your comparison is case insensitive. If 'John' has been used, 'JOHN' should not be accepted. (To do this, you'll need to make a copy of `current_users` containing the lowercase versions of all existing users.)

5-11. Ordinal Numbers: Ordinal numbers indicate their position in a list, such as *1st* or *2nd*. Most ordinal numbers end in *th*, except 1, 2, and 3.

- Store the numbers 1 through 9 in a list.
- Loop through the list.
- Use an if-elif-else chain inside the loop to print the proper ordinal ending for each number. Your output should read "1st 2nd 3rd 4th 5th 6th 7th 8th 9th", and each result should be on a separate line.

Styling Your if Statements

In every example in this chapter, you've seen good styling habits. The only recommendation PEP 8 provides for styling conditional tests is to use a single space around comparison operators, such as `==`, `>=`, `<=`. For example:

```
if age < 4:
```

is better than:

```
if age<4:
```

Such spacing does not affect the way Python interprets your code; it just makes your code easier for you and others to read.

TRY IT YOURSELF

5-12. Styling if statements: Review the programs you wrote in this chapter, and make sure you styled your conditional tests appropriately.

5-13. Your Ideas: At this point, you're a more capable programmer than you were when you started this book. Now that you have a better sense of how real-world situations are modeled in programs, you might be thinking of some problems you could solve with your own programs. Record any new ideas you have about problems you might want to solve as your programming skills continue to improve. Consider games you might want to write, data sets you might want to explore, and web applications you'd like to create.

Summary

In this chapter you learned how to write conditional tests, which always evaluate to True or False. You learned to write simple if statements, if-else chains, and if-elif-else chains. You began using these structures to identify particular conditions you needed to test and to know when those conditions have been met in your programs. You learned to handle certain items in a list differently than all other items while continuing to utilize the efficiency of a for loop. You also revisited Python's style recommendations to ensure that your increasingly complex programs are still relatively easy to read and understand.

In Chapter 6 you'll learn about Python's dictionaries. A dictionary is similar to a list, but it allows you to connect pieces of information. You'll learn to build dictionaries, loop through them, and use them in combination with lists and if statements. Learning about dictionaries will enable you to model an even wider variety of real-world situations.

6

DICTIONARIES



In this chapter you'll learn how to use Python's dictionaries, which allow you to connect pieces of related information. You'll learn how to access the information once it's in a dictionary and how to modify that information. Because dictionaries can store an almost limitless amount of information, I'll show you how to loop through the data in a dictionary. Additionally, you'll learn to nest dictionaries inside lists, lists inside dictionaries, and even dictionaries inside other dictionaries.

Understanding dictionaries allows you to model a variety of real-world objects more accurately. You'll be able to create a dictionary representing a person and then store as much information as you want about that person. You can store their name, age, location, profession, and any other aspect of a person you can describe. You'll be able to store any two kinds of

information that can be matched up, such as a list of words and their meanings, a list of people's names and their favorite numbers, a list of mountains and their elevations, and so forth.

A Simple Dictionary

Consider a game featuring aliens that can have different colors and point values. This simple dictionary stores information about a particular alien:

```
alien.py
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

The dictionary `alien_0` stores the alien's color and point value. The last two lines access and display that information, as shown here:

```
green
5
```

As with most new programming concepts, using dictionaries takes practice. Once you've worked with dictionaries for a bit you'll soon see how effectively they can model real-world situations.

Working with Dictionaries

A *dictionary* in Python is a collection of *key-value pairs*. Each *key* is connected to a value, and you can use a key to access the value associated with that key. A key's value can be a number, a string, a list, or even another dictionary. In fact, you can use any object that you can create in Python as a value in a dictionary.

In Python, a dictionary is wrapped in braces, {}, with a series of key-value pairs inside the braces, as shown in the earlier example:

```
alien_0 = {'color': 'green', 'points': 5}
```

A *key-value pair* is a set of values associated with each other. When you provide a key, Python returns the value associated with that key. Every key is connected to its value by a colon, and individual key-value pairs are separated by commas. You can store as many key-value pairs as you want in a dictionary.

The simplest dictionary has exactly one key-value pair, as shown in this modified version of the `alien_0` dictionary:

```
alien_0 = {'color': 'green'}
```

This dictionary stores one piece of information about alien_0, namely the alien's color. The string 'color' is a key in this dictionary, and its associated value is 'green'.

Accessing Values in a Dictionary

To get the value associated with a key, give the name of the dictionary and then place the key inside a set of square brackets, as shown here:

alien.py

```
alien_0 = {'color': 'green'}  
print(alien_0['color'])
```

This returns the value associated with the key 'color' from the dictionary alien_0:

green

You can have an unlimited number of key-value pairs in a dictionary. For example, here's the original alien_0 dictionary with two key-value pairs:

```
alien_0 = {'color': 'green', 'points': 5}
```

Now you can access either the color or the point value of alien_0. If a player shoots down this alien, you can look up how many points they should earn using code like this:

```
alien_0 = {'color': 'green', 'points': 5}
```

```
❶ new_points = alien_0['points']  
❷ print(f"You just earned {new_points} points!")
```

Once the dictionary has been defined, the code at ❶ pulls the value associated with the key 'points' from the dictionary. This value is then assigned to the variable new_points. The line at ❷ prints a statement about how many points the player just earned:

You just earned 5 points!

If you run this code every time an alien is shot down, the alien's point value will be retrieved.

Adding New Key-Value Pairs

Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time. For example, to add a new key-value pair, you would give the name of the dictionary followed by the new key in square brackets along with the new value.

Let's add two new pieces of information to the alien_0 dictionary: the alien's x- and y-coordinates, which will help us display the alien in a

particular position on the screen. Let's place the alien on the left edge of the screen, 25 pixels down from the top. Because screen coordinates usually start at the upper-left corner of the screen, we'll place the alien on the left edge of the screen by setting the x-coordinate to 0 and 25 pixels from the top by setting its y-coordinate to positive 25, as shown here:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

❶ alien_0['x_position'] = 0
❷ alien_0['y_position'] = 25
print(alien_0)
```

We start by defining the same dictionary that we've been working with. We then print this dictionary, displaying a snapshot of its information. At ❶ we add a new key-value pair to the dictionary: key 'x_position' and value 0. We do the same for key 'y_position' at ❷. When we print the modified dictionary, we see the two additional key-value pairs:

```
{'color': 'green', 'points': 5}
{'color': 'green', 'points': 5, 'y_position': 25, 'x_position': 0}
```

The final version of the dictionary contains four key-value pairs. The original two specify color and point value, and two more specify the alien's position.

NOTE

As of Python 3.7, dictionaries retain the order in which they were defined. When you print a dictionary or loop through its elements, you will see the elements in the same order in which they were added to the dictionary.

Starting with an Empty Dictionary

It's sometimes convenient, or even necessary, to start with an empty dictionary and then add each new item to it. To start filling an empty dictionary, define a dictionary with an empty set of braces and then add each key-value pair on its own line. For example, here's how to build the `alien_0` dictionary using this approach:

alien.py

```
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Here we define an empty `alien_0` dictionary, and then add color and point values to it. The result is the dictionary we've been using in previous examples:

```
{'color': 'green', 'points': 5}
```

Typically, you'll use empty dictionaries when storing user-supplied data in a dictionary or when you write code that generates a large number of key-value pairs automatically.

Modifying Values in a Dictionary

To modify a value in a dictionary, give the name of the dictionary with the key in square brackets and then the new value you want associated with that key. For example, consider an alien that changes from green to yellow as a game progresses:

alien.py

```
alien_0 = {'color': 'green'}
print(f"The alien is {alien_0['color']}.")

alien_0['color'] = 'yellow'
print(f"The alien is now {alien_0['color']}.)")
```

We first define a dictionary for `alien_0` that contains only the alien's color; then we change the value associated with the key 'color' to 'yellow'. The output shows that the alien has indeed changed from green to yellow:

```
The alien is green.
The alien is now yellow.
```

For a more interesting example, let's track the position of an alien that can move at different speeds. We'll store a value representing the alien's current speed and then use it to determine how far to the right the alien should move:

```
alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}
print(f"Original position: {alien_0['x_position']}")

# Move the alien to the right.
# Determine how far to move the alien based on its current speed.
❶ if alien_0['speed'] == 'slow':
    x_increment = 1
elif alien_0['speed'] == 'medium':
    x_increment = 2
else:
    # This must be a fast alien.
    x_increment = 3

    # The new position is the old position plus the increment.
❷ alien_0['x_position'] = alien_0['x_position'] + x_increment

print(f"New position: {alien_0['x_position']}")
```

We start by defining an alien with an initial `x` position and `y` position, and a speed of 'medium'. We've omitted the color and point values for the

sake of simplicity, but this example would work the same way if you included those key-value pairs as well. We also print the original value of `x_position` to see how far the alien moves to the right.

At ❶, an `if-elif-else` chain determines how far the alien should move to the right and assigns this value to the variable `x_increment`. If the alien's speed is '`slow`', it moves one unit to the right; if the speed is '`medium`', it moves two units to the right; and if it's '`fast`', it moves three units to the right. Once the increment has been calculated, it's added to the value of `x_position` at ❷, and the result is stored in the dictionary's `x_position`.

Because this is a medium-speed alien, its position shifts two units to the right:

```
Original x-position: 0
New x-position: 2
```

This technique is pretty cool: by changing one value in the alien's dictionary, you can change the overall behavior of the alien. For example, to turn this medium-speed alien into a fast alien, you would add the line:

```
alien_0['speed'] = 'fast'
```

The `if-elif-else` block would then assign a larger value to `x_increment` the next time the code runs.

Removing Key-Value Pairs

When you no longer need a piece of information that's stored in a dictionary, you can use the `del` statement to completely remove a key-value pair. All `del` needs is the name of the dictionary and the key that you want to remove.

For example, let's remove the key '`points`' from the `alien_0` dictionary along with its value:

```
alien.py    alien_0 = {'color': 'green', 'points': 5}
              print(alien_0)
```

```
❶ del alien_0['points']
      print(alien_0)
```

The line at ❶ tells Python to delete the key '`points`' from the dictionary `alien_0` and to remove the value associated with that key as well. The output shows that the key '`points`' and its value of 5 are deleted from the dictionary, but the rest of the dictionary is unaffected:

```
{'color': 'green', 'points': 5}
{'color': 'green'}
```

NOTE

Be aware that the deleted key-value pair is removed permanently.

A Dictionary of Similar Objects

The previous example involved storing different kinds of information about one object, an alien in a game. You can also use a dictionary to store one kind of information about many objects. For example, say you want to poll a number of people and ask them what their favorite programming language is. A dictionary is useful for storing the results of a simple poll, like this:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}
```

As you can see, we've broken a larger dictionary into several lines. Each key is the name of a person who responded to the poll, and each value is their language choice. When you know you'll need more than one line to define a dictionary, press ENTER after the opening brace. Then indent the next line one level (four spaces), and write the first key-value pair, followed by a comma. From this point forward when you press ENTER, your text editor should automatically indent all subsequent key-value pairs to match the first key-value pair.

Once you've finished defining the dictionary, add a closing brace on a new line after the last key-value pair and indent it one level so it aligns with the keys in the dictionary. It's good practice to include a comma after the last key-value pair as well, so you're ready to add a new key-value pair on the next line.

NOTE

Most editors have some functionality that helps you format extended lists and dictionaries in a similar manner to this example. Other acceptable ways to format long dictionaries are available as well, so you may see slightly different formatting in your editor, or in other sources.

To use this dictionary, given the name of a person who took the poll, you can easily look up their favorite language:

```
favorite_languages.py
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}
```

```
❶ language = favorite_languages['sarah'].title()
print(f"Sarah's favorite language is {language}.")
```

To see which language Sarah chose, we ask for the value at:

```
favorite_languages['sarah']
```

We use this syntax to pull Sarah's favorite language from the dictionary at ❶ and assign it to the variable `language`. Creating a new variable here makes for a much cleaner `print()` call. The output shows Sarah's favorite language:

```
Sarah's favorite language is C.
```

You could use this same syntax with any individual represented in the dictionary.

Using `get()` to Access Values

Using keys in square brackets to retrieve the value you're interested in from a dictionary might cause one potential problem: if the key you ask for doesn't exist, you'll get an error.

Let's see what happens when you ask for the point value of an alien that doesn't have a point value set:

```
alien_no_points.py
```

```
alien_0 = {'color': 'green', 'speed': 'slow'}
print(alien_0['points'])
```

This results in a traceback, showing a `KeyError`:

```
Traceback (most recent call last):
  File "alien_no_points.py", line 2, in <module>
    print(alien_0['points'])
KeyError: 'points'
```

You'll learn more about how to handle errors like this in general in Chapter 10. For dictionaries, specifically, you can use the `get()` method to set a default value that will be returned if the requested key doesn't exist.

The `get()` method requires a key as a first argument. As a second optional argument, you can pass the value to be returned if the key doesn't exist:

```
alien_0 = {'color': 'green', 'speed': 'slow'}

point_value = alien_0.get('points', 'No point value assigned.')
print(point_value)
```

If the key `'points'` exists in the dictionary, you'll get the corresponding value. If it doesn't, you get the default value. In this case, `points` doesn't exist, and we get a clean message instead of an error:

```
No point value assigned.
```

If there's a chance the key you're asking for might not exist, consider using the `get()` method instead of the square bracket notation.

NOTE

If you leave out the second argument in the call to `get()` and the key doesn't exist, Python will return the value `None`. The special value `None` means "no value exists." This is not an error: it's a special value meant to indicate the absence of a value. You'll see more uses for `None` in Chapter 8.

TRY IT YOURSELF

6-1. Person: Use a dictionary to store information about a person you know.

Store their first name, last name, age, and the city in which they live. You should have keys such as `first_name`, `last_name`, `age`, and `city`. Print each piece of information stored in your dictionary.

6-2. Favorite Numbers: Use a dictionary to store people's favorite numbers.

Think of five names, and use them as keys in your dictionary. Think of a favorite number for each person, and store each as a value in your dictionary. Print each person's name and their favorite number. For even more fun, poll a few friends and get some actual data for your program.

6-3. Glossary: A Python dictionary can be used to model an actual dictionary. However, to avoid confusion, let's call it a glossary.

- Think of five programming words you've learned about in the previous chapters. Use these words as the keys in your glossary, and store their meanings as values.
- Print each word and its meaning as neatly formatted output. You might print the word followed by a colon and then its meaning, or print the word on one line and then print its meaning indented on a second line. Use the newline character (`\n`) to insert a blank line between each word-meaning pair in your output.

Looping Through a Dictionary

A single Python dictionary can contain just a few key-value pairs or millions of pairs. Because a dictionary can contain large amounts of data, Python lets you loop through a dictionary. Dictionaries can be used to store information in a variety of ways; therefore, several different ways exist to loop through them. You can loop through all of a dictionary's key-value pairs, through its keys, or through its values.

Looping Through All Key-Value Pairs

Before we explore the different approaches to looping, let's consider a new dictionary designed to store information about a user on a website. The

following dictionary would store one person's username, first name, and last name:

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
}
```

You can access any single piece of information about `user_0` based on what you've already learned in this chapter. But what if you wanted to see everything stored in this user's dictionary? To do so, you could loop through the dictionary using a `for` loop:

```
user.py      user_0 = {
                'username': 'efermi',
                'first': 'enrico',
                'last': 'fermi',
}

❶ for key, value in user_0.items():
❷     print(f"\nKey: {key}")
❸     print(f"Value: {value}")
```

As shown at ❶, to write a `for` loop for a dictionary, you create names for the two variables that will hold the key and value in each key-value pair. You can choose any names you want for these two variables. This code would work just as well if you had used abbreviations for the variable names, like this:

```
for k, v in user_0.items()
```

The second half of the `for` statement at ❶ includes the name of the dictionary followed by the method `items()`, which returns a list of key-value pairs. The `for` loop then assigns each of these pairs to the two variables provided. In the preceding example, we use the variables to print each key ❷, followed by the associated value ❸. The "`\n`" in the first `print()` call ensures that a blank line is inserted before each key-value pair in the output:

```
Key: last
Value: fermi
```

```
Key: first
Value: enrico
```

```
Key: username
Value: efermi
```

Looping through all key-value pairs works particularly well for dictionaries like the *favorite_languages.py* example on page 97, which stores the same kind of information for many different keys. If you loop through the `favorite_languages` dictionary, you get the name of each person in the dictionary and their favorite programming language. Because the keys always refer to a person's name and the value is always a language, we'll use the variables `name` and `language` in the loop instead of `key` and `value`. This will make it easier to follow what's happening inside the loop:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}
```

```
❶ for name, language in favorite_languages.items():
❷     print(f"{name.title()}\'s favorite language is {language.title()}.")
```

The code at ❶ tells Python to loop through each key-value pair in the dictionary. As it works through each pair the key is assigned to the variable `name`, and the value is assigned to the variable `language`. These descriptive names make it much easier to see what the `print()` call at ❷ is doing.

Now, in just a few lines of code, we can display all of the information from the poll:

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Edward's favorite language is Ruby.
Phil's favorite language is Python.
```

This type of looping would work just as well if our dictionary stored the results from polling a thousand or even a million people.

Looping Through All the Keys in a Dictionary

The `keys()` method is useful when you don't need to work with all of the values in a dictionary. Let's loop through the `favorite_languages` dictionary and print the names of everyone who took the poll:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}
```

```
❶ for name in favorite_languages.keys():
    print(name.title())


---


```

The line at ❶ tells Python to pull all the keys from the dictionary `favorite_languages` and assign them one at a time to the variable `name`. The output shows the names of everyone who took the poll:

```
Jen  
Sarah  
Edward  
Phil
```

Looping through the keys is actually the default behavior when looping through a dictionary, so this code would have exactly the same output if you wrote . . .

```
for name in favorite_languages:
```

rather than . . .

```
for name in favorite_languages.keys():
```

You can choose to use the `keys()` method explicitly if it makes your code easier to read, or you can omit it if you wish.

You can access the value associated with any key you care about inside the loop by using the current key. Let's print a message to a couple of friends about the languages they chose. We'll loop through the names in the dictionary as we did previously, but when the name matches one of our friends, we'll display a message about their favorite language:

```
favorite_languages = {  
    --snip--  
}  
  
❶ friends = ['phil', 'sarah']  
for name in favorite_languages.keys():  
    print(name.title())  
  
❷     if name in friends:  
❸         language = favorite_languages[name].title()  
         print(f"\t{name.title()}, I see you love {language}!")
```

At ❶ we make a list of friends that we want to print a message to. Inside the loop, we print each person's name. Then at ❷ we check whether the name we're working with is in the list `friends`. If it is, we determine the person's favorite language using the name of the dictionary and the current value of `name` as the key ❸. We then print a special greeting, including a reference to their language of choice.

Everyone's name is printed, but our friends receive a special message:

```
Hi Jen.  
Hi Sarah.  
    Sarah, I see you love C!  
Hi Edward.
```

Hi Phil.

Phil, I see you love Python!

You can also use the `keys()` method to find out if a particular person was polled. This time, let's find out if Erin took the poll:

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'ruby',  
    'phil': 'python',  
}
```

```
❶ if 'erin' not in favorite_languages.keys():  
    print("Erin, please take our poll!")
```

The `keys()` method isn't just for looping: it actually returns a list of all the keys, and the line at ❶ simply checks if 'erin' is in this list. Because she's not, a message is printed inviting her to take the poll:

Erin, please take our poll!

Looping Through a Dictionary's Keys in a Particular Order

Starting in Python 3.7, looping through a dictionary returns the items in the same order they were inserted. Sometimes, though, you'll want to loop through a dictionary in a different order.

One way to do this is to sort the keys as they're returned in the `for` loop. You can use the `sorted()` function to get a copy of the keys in order:

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'ruby',  
    'phil': 'python',  
}  
  
for name in sorted(favorite_languages.keys()):  
    print(f"{name.title()}, thank you for taking the poll.")
```

This `for` statement is like other `for` statements except that we've wrapped the `sorted()` function around the `dictionary.keys()` method. This tells Python to list all keys in the dictionary and sort that list before looping through it. The output shows everyone who took the poll, with the names displayed in order:

Edward, thank you for taking the poll.
Jen, thank you for taking the poll.
Phil, thank you for taking the poll.
Sarah, thank you for taking the poll.

Looping Through All Values in a Dictionary

If you are primarily interested in the values that a dictionary contains, you can use the `values()` method to return a list of values without any keys. For example, say we simply want a list of all languages chosen in our programming language poll without the name of the person who chose each language:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}

print("The following languages have been mentioned:")
for language in favorite_languages.values():
    print(language.title())
```

The `for` statement here pulls each value from the dictionary and assigns it to the variable `language`. When these values are printed, we get a list of all chosen languages:

```
The following languages have been mentioned:
Python
C
Python
Ruby
```

This approach pulls all the values from the dictionary without checking for repeats. That might work fine with a small number of values, but in a poll with a large number of respondents, this would result in a very repetitive list. To see each language chosen without repetition, we can use a set. A *set* is a collection in which each item must be unique:

```
favorite_languages = {
    --snip--
}

print("The following languages have been mentioned:")
❶ for language in set(favorite_languages.values()):
    print(language.title())
```

When you wrap `set()` around a list that contains duplicate items, Python identifies the unique items in the list and builds a set from those items. At ❶ we use `set()` to pull out the unique languages in `favorite_languages.values()`.

The result is a nonrepetitive list of languages that have been mentioned by people taking the poll:

```
The following languages have been mentioned:
Python
```

As you continue learning about Python, you'll often find a built-in feature of the language that helps you do exactly what you want with your data.

NOTE

You can build a set directly using braces and separating the elements with commas:

```
>>> languages = {'python', 'ruby', 'python', 'c'}
>>> languages
{'ruby', 'python', 'c'}
```

It's easy to mistake sets for dictionaries because they're both wrapped in braces. When you see braces but no key-value pairs, you're probably looking at a set. Unlike lists and dictionaries, sets do not retain items in any specific order.

TRY IT YOURSELF

6-4. Glossary 2: Now that you know how to loop through a dictionary, clean up the code from Exercise 6-3 (page 99) by replacing your series of `print()` calls with a loop that runs through the dictionary's keys and values. When you're sure that your loop works, add five more Python terms to your glossary. When you run your program again, these new words and meanings should automatically be included in the output.

6-5. Rivers: Make a dictionary containing three major rivers and the country each river runs through. One key-value pair might be 'nile': 'egypt'.

- Use a loop to print a sentence about each river, such as *The Nile runs through Egypt.*
- Use a loop to print the name of each river included in the dictionary.
- Use a loop to print the name of each country included in the dictionary.

6-6. Polling: Use the code in `favorite_languages.py` (page 97).

- Make a list of people who should take the favorite languages poll. Include some names that are already in the dictionary and some that are not.
- Loop through the list of people who should take the poll. If they have already taken the poll, print a message thanking them for responding. If they have not yet taken the poll, print a message inviting them to take the poll.

Nesting

Sometimes you'll want to store multiple dictionaries in a list, or a list of items as a value in a dictionary. This is called *nesting*. You can nest dictionaries inside a list, a list of items inside a dictionary, or even a dictionary inside another dictionary. Nesting is a powerful feature, as the following examples will demonstrate.

A List of Dictionaries

The alien_0 dictionary contains a variety of information about one alien, but it has no room to store information about a second alien, much less a screen full of aliens. How can you manage a fleet of aliens? One way is to make a list of aliens in which each alien is a dictionary of information about that alien. For example, the following code builds a list of three aliens:

aliens.py

```
alien_0 = {'color': 'green', 'points': 5}
alien_1 = {'color': 'yellow', 'points': 10}
alien_2 = {'color': 'red', 'points': 15}

❶ aliens = [alien_0, alien_1, alien_2]

for alien in aliens:
    print(alien)
```

We first create three dictionaries, each representing a different alien. At ❶ we store each of these dictionaries in a list called `aliens`. Finally, we loop through the list and print out each alien:

```
{'color': 'green', 'points': 5}
{'color': 'yellow', 'points': 10}
{'color': 'red', 'points': 15}
```

A more realistic example would involve more than three aliens with code that automatically generates each alien. In the following example we use `range()` to create a fleet of 30 aliens:

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
❶ for alien_number in range(30):
❷     new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
❸     aliens.append(new_alien)

# Show the first 5 aliens.
❹ for alien in aliens[:5]:
    print(alien)
    print("...")

# Show how many aliens have been created.
❺ print(f"Total number of aliens: {len(aliens)})")
```

This example begins with an empty list to hold all of the aliens that will be created. At ❶ `range()` returns a series of numbers, which just tells Python how many times we want the loop to repeat. Each time the loop runs we create a new alien ❷ and then append each new alien to the list `aliens` ❸. At ❹ we use a slice to print the first five aliens, and then at ❽ we print the length of the list to prove we've actually generated the full fleet of 30 aliens:

```
{'speed': 'slow', 'color': 'green', 'points': 5}
...
Total number of aliens: 30
```

These aliens all have the same characteristics, but Python considers each one a separate object, which allows us to modify each alien individually.

How might you work with a group of aliens like this? Imagine that one aspect of a game has some aliens changing color and moving faster as the game progresses. When it's time to change colors, we can use a `for` loop and an `if` statement to change the color of aliens. For example, to change the first three aliens to yellow, medium-speed aliens worth 10 points each, we could do this:

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
for alien_number in range(30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10

# Show the first 5 aliens.
for alien in aliens[:5]:
    print(alien)
print("...")
```

Because we want to modify the first three aliens, we loop through a slice that includes only the first three aliens. All of the aliens are green now but that won't always be the case, so we write an `if` statement to make sure

we're only modifying green aliens. If the alien is green, we change the color to 'yellow', the speed to 'medium', and the point value to 10, as shown in the following output:

```
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
...

```

You could expand this loop by adding an `elif` block that turns yellow aliens into red, fast-moving ones worth 15 points each. Without showing the entire program again, that loop would look like this:

```
for alien in aliens[0:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10
    elif alien['color'] == 'yellow':
        alien['color'] = 'red'
        alien['speed'] = 'fast'
        alien['points'] = 15

```

It's common to store a number of dictionaries in a list when each dictionary contains many kinds of information about one object. For example, you might create a dictionary for each user on a website, as we did in `user.py` on page 100, and store the individual dictionaries in a list called `users`. All of the dictionaries in the list should have an identical structure so you can loop through the list and work with each dictionary object in the same way.

A List in a Dictionary

Rather than putting a dictionary inside a list, it's sometimes useful to put a list inside a dictionary. For example, consider how you might describe a pizza that someone is ordering. If you were to use only a list, all you could really store is a list of the pizza's toppings. With a dictionary, a list of toppings can be just one aspect of the pizza you're describing.

In the following example, two kinds of information are stored for each pizza: a type of crust and a list of toppings. The list of toppings is a value associated with the key 'toppings'. To use the items in the list, we give the name of the dictionary and the key 'toppings', as we would any value in the dictionary. Instead of returning a single value, we get a list of toppings:

```
pizza.py  # Store information about a pizza being ordered.
❶ pizza = {
            'crust': 'thick',
            'toppings': ['mushrooms', 'extra cheese'],
        }
```

```
# Summarize the order.  
❷ print(f"You ordered a {pizza['crust']}‑crust pizza "  
      "with the following toppings:")  
  
❸ for topping in pizza['toppings']:  
    print("\t" + topping)
```

We begin at ❶ with a dictionary that holds information about a pizza that has been ordered. One key in the dictionary is 'crust', and the associated value is the string 'thick'. The next key, 'toppings', has a list as its value that stores all requested toppings. At ❷ we summarize the order before building the pizza. When you need to break up a long line in a `print()` call, choose an appropriate point at which to break the line being printed, and end the line with a quotation mark. Indent the next line, add an opening quotation mark, and continue the string. Python will automatically combine all of the strings it finds inside the parentheses. To print the toppings, we write a `for` loop ❸. To access the list of toppings, we use the key 'toppings', and Python grabs the list of toppings from the dictionary.

The following output summarizes the pizza that we plan to build:

```
You ordered a thick-crust pizza with the following toppings:  
  mushrooms  
  extra cheese
```

You can nest a list inside a dictionary any time you want more than one value to be associated with a single key in a dictionary. In the earlier example of favorite programming languages, if we were to store each person's responses in a list, people could choose more than one favorite language. When we loop through the dictionary, the value associated with each person would be a list of languages rather than a single language. Inside the dictionary's `for` loop, we use another `for` loop to run through the list of languages associated with each person:

```
favorite_languages = {  
    'jen': ['python', 'ruby'],  
    'sarah': ['c'],  
    'edward': ['ruby', 'go'],  
    'phil': ['python', 'haskell'],  
}  
  
❷ for name, languages in favorite_languages.items():  
    print(f"\n{name.title()}'s favorite languages are:")  
    ❸ for language in languages:  
        print(f"\t{language.title()}")
```

As you can see at ❶ the value associated with each name is now a list. Notice that some people have one favorite language and others have

multiple favorites. When we loop through the dictionary at ❷, we use the variable name `languages` to hold each value from the dictionary, because we know that each value will be a list. Inside the main dictionary loop, we use another for loop ❸ to run through each person's list of favorite languages. Now each person can list as many favorite languages as they like:

Jen's favorite languages are:

Python
Ruby

Sarah's favorite languages are:

C

Phil's favorite languages are:

Python
Haskell

Edward's favorite languages are:

Ruby
Go

To refine this program even further, you could include an if statement at the beginning of the dictionary's for loop to see whether each person has more than one favorite language by examining the value of `len(languages)`. If a person has more than one favorite, the output would stay the same. If the person has only one favorite language, you could change the wording to reflect that. For example, you could say Sarah's favorite language is C.

NOTE

You should not nest lists and dictionaries too deeply. If you're nesting items much deeper than what you see in the preceding examples or you're working with someone else's code with significant levels of nesting, most likely a simpler way to solve the problem exists.

A Dictionary in a Dictionary

You can nest a dictionary inside another dictionary, but your code can get complicated quickly when you do. For example, if you have several users for a website, each with a unique username, you can use the usernames as the keys in a dictionary. You can then store information about each user by using a dictionary as the value associated with their username. In the following listing, we store three pieces of information about each user: their first name, last name, and location. We'll access this information by looping through the usernames and the dictionary of information associated with each username:

```
many_users.py
users = {
    'aeinstein': {
        'first': 'albert',
```

```
'last': 'einstein',
'location': 'princeton',
},

'mcurie': {
    'first': 'marie',
    'last': 'curie',
    'location': 'paris',
},
}

❶ for username, user_info in users.items():
❷     print(f"\nUsername: {username}")
❸     full_name = f"{user_info['first']} {user_info['last']}"
     location = user_info['location']

❹     print(f"\tFull name: {full_name.title()}")
     print(f"\tLocation: {location.title()}")
```

We first define a dictionary called `users` with two keys: one each for the usernames '`einstein`' and '`mcurie`'. The value associated with each key is a dictionary that includes each user's first name, last name, and location. At ❶ we loop through the `users` dictionary. Python assigns each key to the variable `username`, and the dictionary associated with each `username` is assigned to the variable `user_info`. Once inside the main dictionary loop, we print the `username` at ❷.

At ❸ we start accessing the inner dictionary. The variable `user_info`, which contains the dictionary of user information, has three keys: '`first`', '`last`', and '`location`'. We use each key to generate a neatly formatted full name and location for each person, and then print a summary of what we know about each user ❹:

```
Username: aeinstein
    Full name: Albert Einstein
    Location: Princeton
```

```
Username: mcurie
    Full name: Marie Curie
    Location: Paris
```

Notice that the structure of each user's dictionary is identical. Although not required by Python, this structure makes nested dictionaries easier to work with. If each user's dictionary had different keys, the code inside the `for` loop would be more complicated.

TRY IT YOURSELF

6-7. People: Start with the program you wrote for Exercise 6-1 (page 99).

Make two new dictionaries representing different people, and store all three dictionaries in a list called `people`. Loop through your list of people. As you loop through the list, print everything you know about each person.

6-8. Pets: Make several dictionaries, where each dictionary represents a different pet. In each dictionary, include the kind of animal and the owner's name. Store these dictionaries in a list called `pets`. Next, loop through your list and as you do, print everything you know about each pet.

6-9. Favorite Places: Make a dictionary called `favorite_places`. Think of three names to use as keys in the dictionary, and store one to three favorite places for each person. To make this exercise a bit more interesting, ask some friends to name a few of their favorite places. Loop through the dictionary, and print each person's name and their favorite places.

6-10. Favorite Numbers: Modify your program from Exercise 6-2 (page 99) so each person can have more than one favorite number. Then print each person's name along with their favorite numbers.

6-11. Cities: Make a dictionary called `cities`. Use the names of three cities as keys in your dictionary. Create a dictionary of information about each city and include the country that the city is in, its approximate population, and one fact about that city. The keys for each city's dictionary should be something like `country`, `population`, and `fact`. Print the name of each city and all of the information you have stored about it.

6-12. Extensions: We're now working with examples that are complex enough that they can be extended in any number of ways. Use one of the example programs from this chapter, and extend it by adding new keys and values, changing the context of the program or improving the formatting of the output.

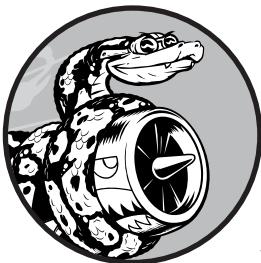
Summary

In this chapter you learned how to define a dictionary and how to work with the information stored in a dictionary. You learned how to access and modify individual elements in a dictionary, and how to loop through all of the information in a dictionary. You learned to loop through a dictionary's key-value pairs, its keys, and its values. You also learned how to nest multiple dictionaries in a list, nest lists in a dictionary, and nest a dictionary inside a dictionary.

In the next chapter you'll learn about `while` loops and how to accept input from people who are using your programs. This will be an exciting chapter, because you'll learn to make all of your programs interactive: they'll be able to respond to user input.

7

USER INPUT AND WHILE LOOPS



Most programs are written to solve an end user's problem. To do so, you usually need to get some information from the user. For a simple example, let's say someone wants to find out whether they're old enough to vote. If you write a program to answer this question, you need to know the user's age before you can provide an answer. The program will need to ask the user to enter, or *input*, their age; once the program has this input, it can compare it to the voting age to determine if the user is old enough and then report the result.

In this chapter you'll learn how to accept user input so your program can then work with it. When your program needs a name, you'll be able to prompt the user for a name. When your program needs a list of names, you'll be able to prompt the user for a series of names. To do this, you'll use the `input()` function.

You'll also learn how to keep programs running as long as users want them to, so they can enter as much information as they need to; then, your program can work with that information. You'll use Python's `while` loop to keep programs running as long as certain conditions remain true.

With the ability to work with user input and the ability to control how long your programs run, you'll be able to write fully interactive programs.

How the `input()` Function Works

The `input()` function pauses your program and waits for the user to enter some text. Once Python receives the user's input, it assigns that input to a variable to make it convenient for you to work with.

For example, the following program asks the user to enter some text, then displays that message back to the user:

```
parrot.py
message = input("Tell me something, and I will repeat it back to you: ")
print(message)
```

The `input()` function takes one argument: the *prompt*, or instructions, that we want to display to the user so they know what to do. In this example, when Python runs the first line, the user sees the prompt `Tell me something, and I will repeat it back to you: .` The program waits while the user enters their response and continues after the user presses ENTER. The response is assigned to the variable `message`, then `print(message)` displays the input back to the user:

```
Tell me something, and I will repeat it back to you: Hello everyone!
Hello everyone!
```

NOTE

Sublime Text and many other editors don't run programs that prompt the user for input. You can use these editors to write programs that prompt for input, but you'll need to run these programs from a terminal. See "Running Python Programs from a Terminal" on page 12.

Writing Clear Prompts

Each time you use the `input()` function, you should include a clear, easy-to-follow prompt that tells the user exactly what kind of information you're looking for. Any statement that tells the user what to enter should work. For example:

```
greeter.py
name = input("Please enter your name: ")
print(f"\nHello, {name}!")
```

Add a space at the end of your prompts (after the colon in the preceding example) to separate the prompt from the user's response and to make it clear to your user where to enter their text. For example:

```
Please enter your name: Eric
Hello, Eric!
```

Sometimes you'll want to write a prompt that's longer than one line. For example, you might want to tell the user why you're asking for certain input. You can assign your prompt to a variable and pass that variable to the `input()` function. This allows you to build your prompt over several lines, then write a clean `input()` statement.

```
greeter.py
prompt = "If you tell us who you are, we can personalize the messages you see."
prompt += "\nWhat is your first name? "

name = input(prompt)
print(f"\nHello, {name}!")
```

This example shows one way to build a multi-line string. The first line assigns the first part of the message to the variable `prompt`. In the second line, the operator `+=` takes the string that was assigned to `prompt` and adds the new string onto the end.

The prompt now spans two lines, again with space after the question mark for clarity:

```
If you tell us who you are, we can personalize the messages you see.
What is your first name? Eric
```

```
Hello, Eric!
```

Using `int()` to Accept Numerical Input

When you use the `input()` function, Python interprets everything the user enters as a string. Consider the following interpreter session, which asks for the user's age:

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age
'21'
```

The user enters the number 21, but when we ask Python for the value of `age`, it returns '`21`', the string representation of the numerical value entered. We know Python interpreted the input as a string because the number is now enclosed in quotes. If all you want to do is print the input, this works well. But if you try to use the input as a number, you'll get an error:

```
>>> age = input("How old are you? ")
How old are you? 21
❶ >>> age >= 18
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
❷ TypeError: unorderable types: str() >= int()
```

When you try to use the input to do a numerical comparison ❶, Python produces an error because it can't compare a string to an integer: the string '21' that's assigned to age can't be compared to the numerical value 18 ❷.

We can resolve this issue by using the `int()` function, which tells Python to treat the input as a numerical value. The `int()` function converts a string representation of a number to a numerical representation, as shown here:

```
>>> age = input("How old are you? ")
How old are you? 21
❶ >>> age = int(age)
>>> age >= 18
True
```

In this example, when we enter 21 at the prompt, Python interprets the number as a string, but the value is then converted to a numerical representation by `int()` ❶. Now Python can run the conditional test: it compares age (which now represents the numerical value 21) and 18 to see if age is greater than or equal to 18. This test evaluates to `True`.

How do you use the `int()` function in an actual program? Consider a program that determines whether people are tall enough to ride a roller coaster:

```
rollercoaster.py
height = input("How tall are you, in inches? ")
height = int(height)

if height >= 48:
    print("\nYou're tall enough to ride!")
else:
    print("\nYou'll be able to ride when you're a little older.")
```

The program can compare `height` to 48 because `height = int(height)` converts the input value to a numerical representation before the comparison is made. If the number entered is greater than or equal to 48, we tell the user that they're tall enough:

```
How tall are you, in inches? 71
```

```
You're tall enough to ride!
```

When you use numerical input to do calculations and comparisons, be sure to convert the input value to a numerical representation first.

The Modulo Operator

A useful tool for working with numerical information is the *modulo operator* (%), which divides one number by another number and returns the remainder:

```
>>> 4 % 3
1
```

```
>>> 5 % 3  
2  
>>> 6 % 3  
0  
>>> 7 % 3  
1
```

The modulo operator doesn't tell you how many times one number fits into another; it just tells you what the remainder is.

When one number is divisible by another number, the remainder is 0, so the modulo operator always returns 0. You can use this fact to determine if a number is even or odd:

```
even_or_odd.py  
number = input("Enter a number, and I'll tell you if it's even or odd: ")  
number = int(number)  
  
if number % 2 == 0:  
    print(f"\nThe number {number} is even.")  
else:  
    print(f"\nThe number {number} is odd.")
```

Even numbers are always divisible by two, so if the modulo of a number and two is zero (here, if `number % 2 == 0`) the number is even. Otherwise, it's odd.

```
Enter a number, and I'll tell you if it's even or odd: 42
```

```
The number 42 is even.
```

TRY IT YOURSELF

7-1. Rental Car: Write a program that asks the user what kind of rental car they would like. Print a message about that car, such as "Let me see if I can find you a Subaru."

7-2. Restaurant Seating: Write a program that asks the user how many people are in their dinner group. If the answer is more than eight, print a message saying they'll have to wait for a table. Otherwise, report that their table is ready.

7-3. Multiples of Ten: Ask the user for a number, and then report whether the number is a multiple of 10 or not.

Introducing while Loops

The `for` loop takes a collection of items and executes a block of code once for each item in the collection. In contrast, the `while` loop runs as long as, or *while*, a certain condition is true.

The while Loop in Action

You can use a `while` loop to count up through a series of numbers. For example, the following `while` loop counts from 1 to 5:

counting.py

```
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1
```

In the first line, we start counting from 1 by assigning `current_number` the value 1. The `while` loop is then set to keep running as long as the value of `current_number` is less than or equal to 5. The code inside the loop prints the value of `current_number` and then adds 1 to that value with `current_number += 1`. (The `+=` operator is shorthand for `current_number = current_number + 1`.)

Python repeats the loop as long as the condition `current_number <= 5` is true. Because 1 is less than 5, Python prints 1 and then adds 1, making the current number 2. Because 2 is less than 5, Python prints 2 and adds 1 again, making the current number 3, and so on. Once the value of `current_number` is greater than 5, the loop stops running and the program ends:

```
1
2
3
4
5
```

The programs you use every day most likely contain `while` loops. For example, a game needs a `while` loop to keep running as long as you want to keep playing, and so it can stop running as soon as you ask it to quit. Programs wouldn't be fun to use if they stopped running before we told them to or kept running even after we wanted to quit, so `while` loops are quite useful.

Letting the User Choose When to Quit

We can make the `parrot.py` program run as long as the user wants by putting most of the program inside a `while` loop. We'll define a *quit value* and then keep the program running as long as the user has not entered the quit value:

parrot.py ❶ `prompt = "\nTell me something, and I will repeat it back to you:\n"
prompt += "\nEnter 'quit' to end the program. "`

```
❷ message = ""  
❸ while message != 'quit':  
    message = input(prompt)  
    print(message)
```

At ❶, we define a prompt that tells the user their two options: entering a message or entering the quit value (in this case, 'quit'). Then we set up a variable `message` ❷ to keep track of whatever value the user enters. We define `message` as an empty string, "", so Python has something to check the first time it reaches the `while` line. The first time the program runs and Python reaches the `while` statement, it needs to compare the value of `message` to 'quit', but no user input has been entered yet. If Python has nothing to compare, it won't be able to continue running the program. To solve this problem, we make sure to give `message` an initial value. Although it's just an empty string, it will make sense to Python and allow it to perform the comparison that makes the `while` loop work. This `while` loop ❸ runs as long as the value of `message` is not 'quit'.

The first time through the loop, `message` is just an empty string, so Python enters the loop. At `message = input(prompt)`, Python displays the prompt and waits for the user to enter their input. Whatever they enter is assigned to `message` and printed; then, Python reevaluates the condition in the `while` statement. As long as the user has not entered the word 'quit', the prompt is displayed again and Python waits for more input. When the user finally enters 'quit', Python stops executing the `while` loop and the program ends:

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. Hello everyone!  
Hello everyone!
```

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. Hello again.  
Hello again.
```

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. quit  
quit
```

This program works well, except that it prints the word 'quit' as if it were an actual message. A simple `if` test fixes this:

```
prompt = "\nTell me something, and I will repeat it back to you:"  
prompt += "\nEnter 'quit' to end the program. "  
  
message = ""  
while message != 'quit':  
    message = input(prompt)  
  
    if message != 'quit':  
        print(message)
```

Now the program makes a quick check before displaying the message and only prints the message if it does not match the quit value:

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. Hello everyone!  
Hello everyone!
```

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. Hello again.  
Hello again.
```

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. quit
```

Using a Flag

In the previous example, we had the program perform certain tasks while a given condition was true. But what about more complicated programs in which many different events could cause the program to stop running?

For example, in a game, several different events can end the game. When the player runs out of ships, their time runs out, or the cities they were supposed to protect are all destroyed, the game should end. It needs to end if any one of these events happens. If many possible events might occur to stop the program, trying to test all these conditions in one `while` statement becomes complicated and difficult.

For a program that should run only as long as many conditions are true, you can define one variable that determines whether or not the entire program is active. This variable, called a *flag*, acts as a signal to the program. We can write our programs so they run while the flag is set to True and stop running when any of several events sets the value of the flag to False. As a result, our overall `while` statement needs to check only one condition: whether or not the flag is currently True. Then, all our other tests (to see if an event has occurred that should set the flag to False) can be neatly organized in the rest of the program.

Let's add a flag to *parrot.py* from the previous section. This flag, which we'll call `active` (though you can call it anything), will monitor whether or not the program should continue running:

```
prompt = "\nTell me something, and I will repeat it back to you:  
prompt += "\nEnter 'quit' to end the program. "
```

```
❶ active = True  
❷ while active:  
    message = input(prompt)  
  
    ❸     if message == 'quit':  
        active = False  
    ❹     else:  
        print(message)
```

We set the variable `active` to `True` ❶ so the program starts in an active state. Doing so makes the `while` statement simpler because no comparison is made in the `while` statement itself; the logic is taken care of in other parts of the program. As long as the `active` variable remains `True`, the loop will continue running ❷.

In the `if` statement inside the `while` loop, we check the value of `message` once the user enters their input. If the user enters 'quit' ❸, we set `active` to `False`, and the `while` loop stops. If the user enters anything other than 'quit' ❹, we print their input as a message.

This program has the same output as the previous example where we placed the conditional test directly in the `while` statement. But now that we have a flag to indicate whether the overall program is in an active state, it would be easy to add more tests (such as `elif` statements) for events that should cause `active` to become `False`. This is useful in complicated programs like games in which there may be many events that should each make the program stop running. When any of these events causes the `active` flag to become `False`, the main game loop will exit, a *Game Over* message can be displayed, and the player can be given the option to play again.

Using `break` to Exit a Loop

To exit a `while` loop immediately without running any remaining code in the loop, regardless of the results of any conditional test, use the `break` statement. The `break` statement directs the flow of your program; you can use it to control which lines of code are executed and which aren't, so the program only executes code that you want it to, when you want it to.

For example, consider a program that asks the user about places they've visited. We can stop the `while` loop in this program by calling `break` as soon as the user enters the 'quit' value:

`cities.py`

```
prompt = "\nPlease enter the name of a city you have visited:"\nprompt += "\nEnter 'quit' when you are finished."
```

```
❶ while True:\n    city = input(prompt)\n\n    if city == 'quit':\n        break\n    else:\n        print(f"I'd love to go to {city.title()}!")
```

A loop that starts with `while True` ❶ will run forever unless it reaches a `break` statement. The loop in this program continues asking the user to enter the names of cities they've been to until they enter 'quit'. When they enter 'quit', the `break` statement runs, causing Python to exit the loop:

```
Please enter the name of a city you have visited:\n(Enter 'quit' when you are finished.) New York\nI'd love to go to New York!
```

```
Please enter the name of a city you have visited:  
(Enter 'quit' when you are finished.) San Francisco  
I'd love to go to San Francisco!
```

```
Please enter the name of a city you have visited:  
(Enter 'quit' when you are finished.) quit
```

NOTE

You can use the `break` statement in any of Python's loops. For example, you could use `break` to quit a `for` loop that's working through a list or a dictionary.

Using `continue` in a Loop

Rather than breaking out of a loop entirely without executing the rest of its code, you can use the `continue` statement to return to the beginning of the loop based on the result of a conditional test. For example, consider a loop that counts from 1 to 10 but prints only the odd numbers in that range:

counting.py

```
current_number = 0
while current_number < 10:
    current_number += 1
    if current_number % 2 == 0:
        continue
    print(current_number)
```

First we set `current_number` to 0. Because it's less than 10, Python enters the `while` loop. Once inside the loop, we increment the count by 1 at ❶, so `current_number` is 1. The `if` statement then checks the modulo of `current_number` and 2. If the modulo is 0 (which means `current_number` is divisible by 2), the `continue` statement tells Python to ignore the rest of the loop and return to the beginning. If the current number is not divisible by 2, the rest of the loop is executed and Python prints the current number:

```
1
3
5
7
9
```

Avoiding Infinite Loops

Every `while` loop needs a way to stop running so it won't continue to run forever. For example, this counting loop should count from 1 to 5:

counting.py

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

But if you accidentally omit the line `x += 1` (as shown next), the loop will run forever:

```
# This loop runs forever!
x = 1
while x <= 5:
    print(x)
```

Now the value of `x` will start at 1 but never change. As a result, the conditional test `x <= 5` will always evaluate to True and the `while` loop will run forever, printing a series of 1s, like this:

```
1
1
1
1
--snip--
```

Every programmer accidentally writes an infinite `while` loop from time to time, especially when a program's loops have subtle exit conditions. If your program gets stuck in an infinite loop, press `CTRL-C` or just close the terminal window displaying your program's output.

To avoid writing infinite loops, test every `while` loop and make sure the loop stops when you expect it to. If you want your program to end when the user enters a certain input value, run the program and enter that value. If the program doesn't end, scrutinize the way your program handles the value that should cause the loop to exit. Make sure at least one part of the program can make the loop's condition `False` or cause it to reach a `break` statement.

NOTE

Sublime Text and some other editors have an embedded output window. This can make it difficult to stop an infinite loop, and you might have to close the editor to end the loop. Try clicking in the output area of the editor before pressing `CTRL-C`, and you should be able to cancel an infinite loop.

TRY IT YOURSELF

7-4. Pizza Toppings: Write a loop that prompts the user to enter a series of pizza toppings until they enter a 'quit' value. As they enter each topping, print a message saying you'll add that topping to their pizza.

7-5. Movie Tickets: A movie theater charges different ticket prices depending on a person's age. If a person is under the age of 3, the ticket is free; if they are between 3 and 12, the ticket is \$10; and if they are over age 12, the ticket is \$15. Write a loop in which you ask users their age, and then tell them the cost of their movie ticket.

(continued)

7-6. Three Exits: Write different versions of either Exercise 7-4 or Exercise 7-5 that do each of the following at least once:

- Use a conditional test in the while statement to stop the loop.
- Use an active variable to control how long the loop runs.
- Use a break statement to exit the loop when the user enters a 'quit' value.

7-7. Infinity: Write a loop that never ends, and run it. (To end the loop, press CTRL-C or close the window displaying the output.)

Using a while Loop with Lists and Dictionaries

So far, we've worked with only one piece of user information at a time. We received the user's input and then printed the input or a response to it. The next time through the while loop, we'd receive another input value and respond to that. But to keep track of many users and pieces of information, we'll need to use lists and dictionaries with our while loops.

A for loop is effective for looping through a list, but you shouldn't modify a list inside a for loop because Python will have trouble keeping track of the items in the list. To modify a list as you work through it, use a while loop. Using while loops with lists and dictionaries allows you to collect, store, and organize lots of input to examine and report on later.

Moving Items from One List to Another

Consider a list of newly registered but unverified users of a website. After we verify these users, how can we move them to a separate list of confirmed users? One way would be to use a while loop to pull users from the list of unconfirmed users as we verify them and then add them to a separate list of confirmed users. Here's what that code might look like:

```
confirmed
_users.py # Start with users that need to be verified,
# and an empty list to hold confirmed users.
❶ unconfirmed_users = ['alice', 'brian', 'candace']
confirmed_users = []

# Verify each user until there are no more unconfirmed users.
# Move each verified user into the list of confirmed users.
❷ while unconfirmed_users:
❸     current_user = unconfirmed_users.pop()

        print(f"Verifying user: {current_user.title()}")
❹     confirmed_users.append(current_user)
```

```
# Display all confirmed users.
print("\nThe following users have been confirmed:")
for confirmed_user in confirmed_users:
    print(confirmed_user.title())
```

We begin with a list of unconfirmed users at ❶ (Alice, Brian, and Candace) and an empty list to hold confirmed users. The `while` loop at ❷ runs as long as the list `unconfirmed_users` is not empty. Within this loop, the `pop()` function at ❸ removes unverified users one at a time from the end of `unconfirmed_users`. Here, because Candace is last in the `unconfirmed_users` list, her name will be the first to be removed, assigned to `current_user`, and added to the `confirmed_users` list at ❹. Next is Brian, then Alice.

We simulate confirming each user by printing a verification message and then adding them to the list of confirmed users. As the list of unconfirmed users shrinks, the list of confirmed users grows. When the list of unconfirmed users is empty, the loop stops and the list of confirmed users is printed:

```
Verifying user: Candace
Verifying user: Brian
Verifying user: Alice

The following users have been confirmed:
Candace
Brian
Alice
```

Removing All Instances of Specific Values from a List

In Chapter 3 we used `remove()` to remove a specific value from a list. The `remove()` function worked because the value we were interested in appeared only once in the list. But what if you want to remove all instances of a value from a list?

Say you have a list of pets with the value 'cat' repeated several times. To remove all instances of that value, you can run a `while` loop until 'cat' is no longer in the list, as shown here:

```
pets.py
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

We start with a list containing multiple instances of 'cat'. After printing the list, Python enters the `while` loop because it finds the value 'cat' in the list

at least once. Once inside the loop, Python removes the first instance of 'cat', returns to the `while` line, and then reenters the loop when it finds that 'cat' is still in the list. It removes each instance of 'cat' until the value is no longer in the list, at which point Python exits the loop and prints the list again:

```
[ 'dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
[ 'dog', 'dog', 'goldfish', 'rabbit']
```

Filling a Dictionary with User Input

You can prompt for as much input as you need in each pass through a `while` loop. Let's make a polling program in which each pass through the loop prompts for the participant's name and response. We'll store the data we gather in a dictionary, because we want to connect each response with a particular user:

*mountain
_poll.py*

```
responses = {}

# Set a flag to indicate that polling is active.
polling_active = True

while polling_active:
    # Prompt for the person's name and response.
    ❶    name = input("\nWhat is your name? ")
        response = input("Which mountain would you like to climb someday? ")

    # Store the response in the dictionary.
    ❷    responses[name] = response

    # Find out if anyone else is going to take the poll.
    ❸    repeat = input("Would you like to let another person respond? (yes/ no) ")
    if repeat == 'no':
        polling_active = False

# Polling is complete. Show the results.
print("\n--- Poll Results ---")
❹    for name, response in responses.items():
        print(f"{name} would like to climb {response}.")
```

The program first defines an empty dictionary (`responses`) and sets a flag (`polling_active`) to indicate that polling is active. As long as `polling_active` is `True`, Python will run the code in the `while` loop.

Within the loop, the user is prompted to enter their name and a mountain they'd like to climb ❶. That information is stored in the `responses` dictionary ❷, and the user is asked whether or not to keep the poll running ❸. If they enter yes, the program enters the `while` loop again. If they enter no, the `polling_active` flag is set to `False`, the `while` loop stops running, and the final code block at ❹ displays the results of the poll.

If you run this program and enter sample responses, you should see output like this:

What is your name? **Eric**

Which mountain would you like to climb someday? **Denali**

Would you like to let another person respond? (yes/ no) **yes**

What is your name? **Lynn**

Which mountain would you like to climb someday? **Devil's Thumb**

Would you like to let another person respond? (yes/ no) **no**

--- Poll Results ---

Lynn would like to climb Devil's Thumb.

Eric would like to climb Denali.

TRY IT YOURSELF

7-8. Deli: Make a list called `sandwich_orders` and fill it with the names of various sandwiches. Then make an empty list called `finished_sandwiches`. Loop through the list of sandwich orders and print a message for each order, such as I made your tuna sandwich. As each sandwich is made, move it to the list of finished sandwiches. After all the sandwiches have been made, print a message listing each sandwich that was made.

7-9. No Pastrami: Using the list `sandwich_orders` from Exercise 7-8, make sure the sandwich 'pastrami' appears in the list at least three times. Add code near the beginning of your program to print a message saying the deli has run out of pastrami, and then use a while loop to remove all occurrences of 'pastrami' from `sandwich_orders`. Make sure no pastrami sandwiches end up in `finished_sandwiches`.

7-10. Dream Vacation: Write a program that polls users about their dream vacation. Write a prompt similar to *If you could visit one place in the world, where would you go?* Include a block of code that prints the results of the poll.

Summary

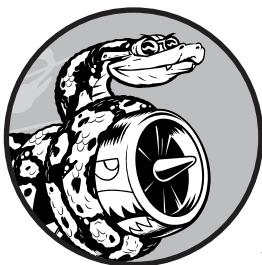
In this chapter you learned how to use `input()` to allow users to provide their own information in your programs. You learned to work with both text and numerical input and how to use `while` loops to make your programs run as long as your users want them to. You saw several ways to control the flow of a `while` loop by setting an active flag, using the `break` statement, and

using the `continue` statement. You learned how to use a `while` loop to move items from one list to another and how to remove all instances of a value from a list. You also learned how `while` loops can be used with dictionaries.

In Chapter 8 you'll learn about *functions*. Functions allow you to break your programs into small parts, each of which does one specific job. You can call a function as many times as you want, and you can store your functions in separate files. By using functions, you'll be able to write more efficient code that's easier to troubleshoot and maintain and that can be reused in many different programs.

8

FUNCTIONS



In this chapter you'll learn to write *functions*, which are named blocks of code that are designed to do one specific job.

When you want to perform a particular task that you've defined in a function, you *call* the function responsible for it. If you need to perform that task multiple times throughout your program, you don't need to type all the code for the same task again and again; you just call the function dedicated to handling that task, and the call tells Python to run the code inside the function. You'll find that using functions makes your programs easier to write, read, test, and fix.

In this chapter you'll also learn ways to pass information to functions. You'll learn how to write certain functions whose primary job is to display information and other functions designed to process data and return a value or set of values. Finally, you'll learn to store functions in separate files called *modules* to help organize your main program files.

Defining a Function

Here's a simple function named `greet_user()` that prints a greeting:

```
greeter.py ❶ def greet_user():
❷     """Display a simple greeting."""
❸     print("Hello!")
❹ greet_user()
```

This example shows the simplest structure of a function. The line at ❶ uses the keyword `def` to inform Python that you're defining a function. This is the *function definition*, which tells Python the name of the function and, if applicable, what kind of information the function needs to do its job. The parentheses hold that information. In this case, the name of the function is `greet_user()`, and it needs no information to do its job, so its parentheses are empty. (Even so, the parentheses are required.) Finally, the definition ends in a colon.

Any indented lines that follow `def greet_user():` make up the *body* of the function. The text at ❷ is a comment called a *docstring*, which describes what the function does. Docstrings are enclosed in triple quotes, which Python looks for when it generates documentation for the functions in your programs.

The line `print("Hello!")` ❸ is the only line of actual code in the body of this function, so `greet_user()` has just one job: `print("Hello!")`.

When you want to use this function, you call it. A *function call* tells Python to execute the code in the function. To *call* a function, you write the name of the function, followed by any necessary information in parentheses, as shown at ❹. Because no information is needed here, calling our function is as simple as entering `greet_user()`. As expected, it prints `Hello!`:

```
Hello!
```

Passing Information to a Function

Modified slightly, the function `greet_user()` can not only tell the user `Hello!` but also greet them by name. For the function to do this, you enter `username` in the parentheses of the function's definition at `def greet_user()`. By adding `username` here you allow the function to accept any value of `username` you specify. The function now expects you to provide a value for `username` each time you call it. When you call `greet_user()`, you can pass it a name, such as '`jesse`', inside the parentheses:

```
def greet_user(username):
    """Display a simple greeting."""
    print(f"Hello, {username.title()}!")

greet_user('jesse')
```

Entering `greet_user('jesse')` calls `greet_user()` and gives the function the information it needs to execute the `print()` call. The function accepts the name you passed it and displays the greeting for that name:

Hello, Jesse!

Likewise, entering `greet_user('sarah')` calls `greet_user()`, passes it 'sarah', and prints Hello, Sarah! You can call `greet_user()` as often as you want and pass it any name you want to produce a predictable output every time.

Arguments and Parameters

In the preceding `greet_user()` function, we defined `greet_user()` to require a value for the variable `username`. Once we called the function and gave it the information (a person's name), it printed the right greeting.

The variable `username` in the definition of `greet_user()` is an example of a *parameter*, a piece of information the function needs to do its job. The value 'jesse' in `greet_user('jesse')` is an example of an *argument*. An argument is a piece of information that's passed from a function call to a function. When we call the function, we place the value we want the function to work with in parentheses. In this case the argument 'jesse' was passed to the function `greet_user()`, and the value was assigned to the parameter `username`.

NOTE

People sometimes speak of arguments and parameters interchangeably. Don't be surprised if you see the variables in a function definition referred to as arguments or the variables in a function call referred to as parameters.

TRY IT YOURSELF

8-1. Message: Write a function called `display_message()` that prints one sentence telling everyone what you are learning about in this chapter. Call the function, and make sure the message displays correctly.

8-2. Favorite Book: Write a function called `favorite_book()` that accepts one parameter, `title`. The function should print a message, such as One of my favorite books is Alice in Wonderland. Call the function, making sure to include a book title as an argument in the function call.

Passing Arguments

Because a function definition can have multiple parameters, a function call may need multiple arguments. You can pass arguments to your functions in a number of ways. You can use *positional arguments*, which need to be in

the same order the parameters were written; *keyword arguments*, where each argument consists of a variable name and a value; and lists and dictionaries of values. Let's look at each of these in turn.

Positional Arguments

When you call a function, Python must match each argument in the function call with a parameter in the function definition. The simplest way to do this is based on the order of the arguments provided. Values matched up this way are called *positional arguments*.

To see how this works, consider a function that displays information about pets. The function tells us what kind of animal each pet is and the pet's name, as shown here:

```
pets.py ❶ def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

❷ describe_pet('hamster', 'harry')
```

The definition shows that this function needs a type of animal and the animal's name ❶. When we call `describe_pet()`, we need to provide an animal type and a name, in that order. For example, in the function call, the argument '`hamster`' is assigned to the parameter `animal_type` and the argument '`harry`' is assigned to the parameter `pet_name` ❷. In the function body, these two parameters are used to display information about the pet being described.

The output describes a hamster named Harry:

```
I have a hamster.
My hamster's name is Harry.
```

Multiple Function Calls

You can call a function as many times as needed. Describing a second, different pet requires just one more call to `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

In this second function call, we pass `describe_pet()` the arguments '`dog`' and '`willie`'. As with the previous set of arguments we used, Python matches '`dog`' with the parameter `animal_type` and '`willie`' with the parameter `pet_name`.

As before, the function does its job, but this time it prints values for a dog named Willie. Now we have a hamster named Harry and a dog named Willie:

I have a hamster.
My hamster's name is Harry.

I have a dog.
My dog's name is Willie.

Calling a function multiple times is a very efficient way to work. The code describing a pet is written once in the function. Then, anytime you want to describe a new pet, you call the function with the new pet's information. Even if the code for describing a pet were to expand to ten lines, you could still describe a new pet in just one line by calling the function again.

You can use as many positional arguments as you need in your functions. Python works through the arguments you provide when calling the function and matches each one with the corresponding parameter in the function's definition.

Order Matters in Positional Arguments

You can get unexpected results if you mix up the order of the arguments in a function call when using positional arguments:

```
def describe_pet(animal_type, pet_name):  
    """Display information about a pet."""  
    print(f"\nI have a {animal_type}.")  
    print(f"My {animal_type}'s name is {pet_name.title()}.")  
  
describe_pet('harry', 'hamster')
```

In this function call we list the name first and the type of animal second. Because the argument 'harry' is listed first this time, that value is assigned to the parameter `animal_type`. Likewise, 'hamster' is assigned to `pet_name`. Now we have a "harry" named "Hamster":

I have a harry.
My harry's name is Hamster.

If you get funny results like this, check to make sure the order of the arguments in your function call matches the order of the parameters in the function's definition.

Keyword Arguments

A *keyword argument* is a name-value pair that you pass to a function. You directly associate the name and the value within the argument, so when you pass the argument to the function, there's no confusion (you won't end up

with a harry named Hamster). Keyword arguments free you from having to worry about correctly ordering your arguments in the function call, and they clarify the role of each value in the function call.

Let's rewrite *pets.py* using keyword arguments to call `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(animal_type='hamster', pet_name='harry')
```

The function `describe_pet()` hasn't changed. But when we call the function, we explicitly tell Python which parameter each argument should be matched with. When Python reads the function call, it knows to assign the argument 'hamster' to the parameter `animal_type` and the argument 'harry' to `pet_name`. The output correctly shows that we have a hamster named Harry.

The order of keyword arguments doesn't matter because Python knows where each value should go. The following two function calls are equivalent:

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

NOTE

When you use keyword arguments, be sure to use the exact names of the parameters in the function's definition.

Default Values

When writing a function, you can define a *default value* for each parameter. If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter's default value. So when you define a default value for a parameter, you can exclude the corresponding argument you'd usually write in the function call. Using default values can simplify your function calls and clarify the ways in which your functions are typically used.

For example, if you notice that most of the calls to `describe_pet()` are being used to describe dogs, you can set the default value of `animal_type` to 'dog'. Now anyone calling `describe_pet()` for a dog can omit that information:

```
def describe_pet(pet_name, animal_type='dog'):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(pet_name='willie')
```

We changed the definition of `describe_pet()` to include a default value, 'dog', for `animal_type`. Now when the function is called with no `animal_type` specified, Python knows to use the value 'dog' for this parameter:

```
I have a dog.  
My dog's name is Willie.
```

Note that the order of the parameters in the function definition had to be changed. Because the default value makes it unnecessary to specify a type of animal as an argument, the only argument left in the function call is the pet's name. Python still interprets this as a positional argument, so if the function is called with just a pet's name, that argument will match up with the first parameter listed in the function's definition. This is the reason the first parameter needs to be `pet_name`.

The simplest way to use this function now is to provide just a dog's name in the function call:

```
describe_pet('willie')
```

This function call would have the same output as the previous example. The only argument provided is 'willie', so it is matched up with the first parameter in the definition, `pet_name`. Because no argument is provided for `animal_type`, Python uses the default value 'dog'.

To describe an animal other than a dog, you could use a function call like this:

```
describe_pet(pet_name='harry', animal_type='hamster')
```

Because an explicit argument for `animal_type` is provided, Python will ignore the parameter's default value.

NOTE

When you use default values, any parameter with a default value needs to be listed after all the parameters that don't have default values. This allows Python to continue interpreting positional arguments correctly.

Equivalent Function Calls

Because positional arguments, keyword arguments, and default values can all be used together, often you'll have several equivalent ways to call a function. Consider the following definition for `describe_pet()` with one default value provided:

```
def describe_pet(pet_name, animal_type='dog'):
```

With this definition, an argument always needs to be provided for `pet_name`, and this value can be provided using the positional or keyword

format. If the animal being described is not a dog, an argument for `animal_type` must be included in the call, and this argument can also be specified using the positional or keyword format.

All of the following calls would work for this function:

```
# A dog named Willie.  
describe_pet('willie')  
describe_pet(pet_name='willie')  
  
# A hamster named Harry.  
describe_pet('harry', 'hamster')  
describe_pet(pet_name='harry', animal_type='hamster')  
describe_pet(animal_type='hamster', pet_name='harry')
```

Each of these function calls would have the same output as the previous examples.

NOTE

It doesn't really matter which calling style you use. As long as your function calls produce the output you want, just use the style you find easiest to understand.

Avoiding Argument Errors

When you start to use functions, don't be surprised if you encounter errors about unmatched arguments. Unmatched arguments occur when you provide fewer or more arguments than a function needs to do its work. For example, here's what happens if we try to call `describe_pet()` with no arguments:

```
def describe_pet(animal_type, pet_name):  
    """Display information about a pet."""  
    print(f"\nI have a {animal_type}.")  
    print(f"My {animal_type}'s name is {pet_name.title()}")  
  
describe_pet()
```

Python recognizes that some information is missing from the function call, and the traceback tells us that:

```
Traceback (most recent call last):  
①  File "pets.py", line 6, in <module>  
②      describe_pet()  
③ TypeError: describe_pet() missing 2 required positional arguments: 'animal_type' and 'pet_name'
```

At ① the traceback tells us the location of the problem, allowing us to look back and see that something went wrong in our function call. At ② the offending function call is written out for us to see. At ③ the traceback

tells us the call is missing two arguments and reports the names of the missing arguments. If this function were in a separate file, we could probably rewrite the call correctly without having to open that file and read the function code.

Python is helpful in that it reads the function's code for us and tells us the names of the arguments we need to provide. This is another motivation for giving your variables and functions descriptive names. If you do, Python's error messages will be more useful to you and anyone else who might use your code.

If you provide too many arguments, you should get a similar traceback that can help you correctly match your function call to the function definition.

TRY IT YOURSELF

8-3. T-Shirt: Write a function called `make_shirt()` that accepts a size and the text of a message that should be printed on the shirt. The function should print a sentence summarizing the size of the shirt and the message printed on it.

Call the function once using positional arguments to make a shirt. Call the function a second time using keyword arguments.

8-4. Large Shirts: Modify the `make_shirt()` function so that shirts are large by default with a message that reads *I love Python*. Make a large shirt and a medium shirt with the default message, and a shirt of any size with a different message.

8-5. Cities: Write a function called `describe_city()` that accepts the name of a city and its country. The function should print a simple sentence, such as Reykjavik is in Iceland. Give the parameter for the country a default value. Call your function for three different cities, at least one of which is not in the default country.

Return Values

A function doesn't always have to display its output directly. Instead, it can process some data and then return a value or set of values. The value the function returns is called a *return value*. The `return` statement takes a value from inside a function and sends it back to the line that called the function. Return values allow you to move much of your program's grunt work into functions, which can simplify the body of your program.

Returning a Simple Value

Let's look at a function that takes a first and last name, and returns a neatly formatted full name:

```
formatted_name.py ❶ def get_formatted_name(first_name, last_name):
                  """Return a full name, neatly formatted."""
❷     full_name = f"{first_name} {last_name}"
❸     return full_name.title()

❹ musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

The definition of `get_formatted_name()` takes as parameters a first and last name ❶. The function combines these two names, adds a space between them, and assigns the result to `full_name` ❷. The value of `full_name` is converted to title case, and then returned to the calling line at ❸.

When you call a function that returns a value, you need to provide a variable that the return value can be assigned to. In this case, the returned value is assigned to the variable `musician` at ❹. The output shows a neatly formatted name made up of the parts of a person's name:

```
Jimi Hendrix
```

This might seem like a lot of work to get a neatly formatted name when we could have just written:

```
print("Jimi Hendrix")
```

But when you consider working with a large program that needs to store many first and last names separately, functions like `get_formatted_name()` become very useful. You store first and last names separately and then call this function whenever you want to display a full name.

Making an Argument Optional

Sometimes it makes sense to make an argument optional so that people using the function can choose to provide extra information only if they want to. You can use default values to make an argument optional.

For example, say we want to expand `get_formatted_name()` to handle middle names as well. A first attempt to include middle names might look like this:

```
def get_formatted_name(first_name, middle_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {middle_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

This function works when given a first, middle, and last name. The function takes in all three parts of a name and then builds a string out of them. The function adds spaces where appropriate and converts the full name to title case:

John Lee Hooker

But middle names aren't always needed, and this function as written would not work if you tried to call it with only a first name and a last name. To make the middle name optional, we can give the `middle_name` argument an empty default value and ignore the argument unless the user provides a value. To make `get_formatted_name()` work without a middle name, we set the default value of `middle_name` to an empty string and move it to the end of the list of parameters:

```
❶ def get_formatted_name(first_name, last_name, middle_name=' '):
    """Return a full name, neatly formatted."""
❷     if middle_name:
        full_name = f'{first_name} {middle_name} {last_name}'
❸     else:
        full_name = f'{first_name} {last_name}'
    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)

❹ musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)
```

In this example, the name is built from three possible parts. Because there's always a first and last name, these parameters are listed first in the function's definition. The middle name is optional, so it's listed last in the definition, and its default value is an empty string ❶.

In the body of the function, we check to see if a middle name has been provided. Python interprets non-empty strings as True, so if `middle_name` evaluates to True if a middle name argument is in the function call ❷. If a middle name is provided, the first, middle, and last names are combined to form a full name. This name is then changed to title case and returned to the function call line where it's assigned to the variable `musician` and printed. If no middle name is provided, the empty string fails the if test and the else block runs ❸. The full name is made with just a first and last name, and the formatted name is returned to the calling line where it's assigned to `musician` and printed.

Calling this function with a first and last name is straightforward. If we're using a middle name, however, we have to make sure the middle name is the last argument passed so Python will match up the positional arguments correctly ❹.

This modified version of our function works for people with just a first and last name, and it works for people who have a middle name as well:

```
Jimi Hendrix  
John Lee Hooker
```

Optional values allow functions to handle a wide range of use cases while letting function calls remain as simple as possible.

Returning a Dictionary

A function can return any kind of value you need it to, including more complicated data structures like lists and dictionaries. For example, the following function takes in parts of a name and returns a dictionary representing a person:

```
person.py    def build_person(first_name, last_name):  
        """Return a dictionary of information about a person."""  
❶      person = {'first': first_name, 'last': last_name}  
❷      return person  
  
❸ musician = build_person('jimi', 'hendrix')  
❹ print(musician)
```

The function `build_person()` takes in a first and last name, and puts these values into a dictionary at ❶. The value of `first_name` is stored with the key '`first`', and the value of `last_name` is stored with the key '`last`'. The entire dictionary representing the person is returned at ❷. The return value is printed at ❸ with the original two pieces of textual information now stored in a dictionary:

```
{'first': 'jimi', 'last': 'hendrix'}
```

This function takes in simple textual information and puts it into a more meaningful data structure that lets you work with the information beyond just printing it. The strings '`jimi`' and '`hendrix`' are now labeled as a first name and last name. You can easily extend this function to accept optional values like a middle name, an age, an occupation, or any other information you want to store about a person. For example, the following change allows you to store a person's age as well:

```
def build_person(first_name, last_name, age=None):  
    """Return a dictionary of information about a person."""  
    person = {'first': first_name, 'last': last_name}  
    if age:  
        person['age'] = age  
    return person  
  
musician = build_person('jimi', 'hendrix', age=27)  
print(musician)
```

We add a new optional parameter `age` to the function definition and assign the parameter the special value `None`, which is used when a variable has no specific value assigned to it. You can think of `None` as a placeholder value. In conditional tests, `None` evaluates to `False`. If the function call includes a value for `age`, that value is stored in the dictionary. This function always stores a person's name, but it can also be modified to store any other information you want about a person.

Using a Function with a `while` Loop

You can use functions with all the Python structures you've learned about so far. For example, let's use the `get_formatted_name()` function with a `while` loop to greet users more formally. Here's a first attempt at greeting people using their first and last names:

greeter.py

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# This is an infinite loop!
while True:
❶    print("\nPlease tell me your name:")
    f_name = input("First name: ")
    l_name = input("Last name: ")

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

For this example, we use a simple version of `get_formatted_name()` that doesn't involve middle names. The `while` loop asks the user to enter their name, and we prompt for their first and last name separately ❶.

But there's one problem with this `while` loop: We haven't defined a quit condition. Where do you put a quit condition when you ask for a series of inputs? We want the user to be able to quit as easily as possible, so each prompt should offer a way to quit. The `break` statement offers a straightforward way to exit the loop at either prompt:

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    f_name = input("First name: ")
    if f_name == 'q':
        break
```

```
l_name = input("Last name: ")
if l_name == 'q':
    break

formatted_name = get_formatted_name(f_name, l_name)
print(f"\nHello, {formatted_name}!")
```

We add a message that informs the user how to quit, and then we break out of the loop if the user enters the quit value at either prompt. Now the program will continue greeting people until someone enters 'q' for either name:

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: eric
Last name: matthes
```

```
Hello, Eric Matthes!
```

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: q
```

TRY IT YOURSELF

8-6. City Names: Write a function called `city_country()` that takes in the name of a city and its country. The function should return a string formatted like this:

```
"Santiago, Chile"
```

Call your function with at least three city-country pairs, and print the values that are returned.

8-7. Album: Write a function called `make_album()` that builds a dictionary describing a music album. The function should take in an artist name and an album title, and it should return a dictionary containing these two pieces of information. Use the function to make three dictionaries representing different albums. Print each return value to show that the dictionaries are storing the album information correctly.

Use `None` to add an optional parameter to `make_album()` that allows you to store the number of songs on an album. If the calling line includes a value for the number of songs, add that value to the album's dictionary. Make at least one new function call that includes the number of songs on an album.

8-8. User Albums: Start with your program from Exercise 8-7. Write a while loop that allows users to enter an album's artist and title. Once you have that information, call `make_album()` with the user's input and print the dictionary that's created. Be sure to include a quit value in the while loop.

Passing a List

You'll often find it useful to pass a list to a function, whether it's a list of names, numbers, or more complex objects, such as dictionaries. When you pass a list to a function, the function gets direct access to the contents of the list. Let's use functions to make working with lists more efficient.

Say we have a list of users and want to print a greeting to each. The following example sends a list of names to a function called `greet_users()`, which greets each person in the list individually:

```
greet_users.py
def greet_users(names):
    """Print a simple greeting to each user in the list."""
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)

❶ usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

We define `greet_users()` so it expects a list of names, which it assigns to the parameter `names`. The function loops through the list it receives and prints a greeting to each user. At ❶ we define a list of users and then pass the list `usernames` to `greet_users()` in our function call:

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

This is the output we wanted. Every user sees a personalized greeting, and you can call the function any time you want to greet a specific set of users.

Modifying a List in a Function

When you pass a list to a function, the function can modify the list. Any changes made to the list inside the function's body are permanent, allowing you to work efficiently even when you're dealing with large amounts of data.

Consider a company that creates 3D printed models of designs that users submit. Designs that need to be printed are stored in a list, and after being printed they're moved to a separate list. The following code does this without using functions:

```
printing_models.py
# Start with some designs that need to be printed.
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

# Simulate printing each design, until none are left.
# Move each design to completed_models after printing.
while unprinted_designs:
    current_design = unprinted_designs.pop()
```

```
print(f"Printing model: {current_design}")
completed_models.append(current_design)

# Display all completed models.
print("\nThe following models have been printed:")
for completed_model in completed_models:
    print(completed_model)
```

This program starts with a list of designs that need to be printed and an empty list called `completed_models` that each design will be moved to after it has been printed. As long as designs remain in `unprinted_designs`, the while loop simulates printing each design by removing a design from the end of the list, storing it in `current_design`, and displaying a message that the current design is being printed. It then adds the design to the list of completed models. When the loop is finished running, a list of the designs that have been printed is displayed:

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: phone case
```

```
The following models have been printed:
dodecahedron
robot pendant
phone case
```

We can reorganize this code by writing two functions, each of which does one specific job. Most of the code won't change; we're just making it more carefully structured. The first function will handle printing the designs, and the second will summarize the prints that have been made:

```
❶ def print_models(unprinted_designs, completed_models):
    """
    Simulate printing each design, until none are left.
    Move each design to completed_models after printing.
    """
    while unprinted_designs:
        current_design = unprinted_designs.pop()
        print(f"Printing model: {current_design}")
        completed_models.append(current_design)

❷ def show_completed_models(completed_models):
    """
    Show all the models that were printed.
    """
    print("\nThe following models have been printed:")
    for completed_model in completed_models:
        print(completed_model)

unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

At ❶ we define the function `print_models()` with two parameters: a list of designs that need to be printed and a list of completed models. Given these two lists, the function simulates printing each design by emptying the list of unprinted designs and filling up the list of completed models. At ❷ we define the function `show_completed_models()` with one parameter: the list of completed models. Given this list, `show_completed_models()` displays the name of each model that was printed.

This program has the same output as the version without functions, but the code is much more organized. The code that does most of the work has been moved to two separate functions, which makes the main part of the program easier to understand. Look at the body of the program to see how much easier it is to understand what this program is doing:

```
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

We set up a list of unprinted designs and an empty list that will hold the completed models. Then, because we've already defined our two functions, all we have to do is call them and pass them the right arguments. We call `print_models()` and pass it the two lists it needs; as expected, `print_models()` simulates printing the designs. Then we call `show_completed_models()` and pass it the list of completed models so it can report the models that have been printed. The descriptive function names allow others to read this code and understand it, even without comments.

This program is easier to extend and maintain than the version without functions. If we need to print more designs later on, we can simply call `print_models()` again. If we realize the printing code needs to be modified, we can change the code once, and our changes will take place everywhere the function is called. This technique is more efficient than having to update code separately in several places in the program.

This example also demonstrates the idea that every function should have one specific job. The first function prints each design, and the second displays the completed models. This is more beneficial than using one function to do both jobs. If you're writing a function and notice the function is doing too many different tasks, try to split the code into two functions. Remember that you can always call a function from another function, which can be helpful when splitting a complex task into a series of steps.

Preventing a Function from Modifying a List

Sometimes you'll want to prevent a function from modifying a list. For example, say that you start with a list of unprinted designs and write a function to move them to a list of completed models, as in the previous example. You may decide that even though you've printed all the designs, you want to keep the original list of unprinted designs for your records.

But because you moved all the design names out of `unprinted_designs`, the list is now empty, and the empty list is the only version you have; the original is gone. In this case, you can address this issue by passing the function a copy of the list, not the original. Any changes the function makes to the list will affect only the copy, leaving the original list intact.

You can send a copy of a list to a function like this:

```
function_name(list_name[:])
```

The slice notation `[:]` makes a copy of the list to send to the function. If we didn't want to empty the list of unprinted designs in `printing_models.py`, we could call `print_models()` like this:

```
print_models(unprinted_designs[:], completed_models)
```

The function `print_models()` can do its work because it still receives the names of all unprinted designs. But this time it uses a copy of the original unprinted designs list, not the actual `unprinted_designs` list. The list `completed_models` will fill up with the names of printed models like it did before, but the original list of unprinted designs will be unaffected by the function.

Even though you can preserve the contents of a list by passing a copy of it to your functions, you should pass the original list to functions unless you have a specific reason to pass a copy. It's more efficient for a function to work with an existing list to avoid using the time and memory needed to make a separate copy, especially when you're working with large lists.

TRY IT YOURSELF

8-9. Messages: Make a list containing a series of short text messages. Pass the list to a function called `show_messages()`, which prints each text message.

8-10. Sending Messages: Start with a copy of your program from Exercise 8-9. Write a function called `send_messages()` that prints each text message and moves each message to a new list called `sent_messages` as it's printed. After calling the function, print both of your lists to make sure the messages were moved correctly.

8-11. Archived Messages: Start with your work from Exercise 8-10. Call the function `send_messages()` with a copy of the list of messages. After calling the function, print both of your lists to show that the original list has retained its messages.

Passing an Arbitrary Number of Arguments

Sometimes you won't know ahead of time how many arguments a function needs to accept. Fortunately, Python allows a function to collect an arbitrary number of arguments from the calling statement.

For example, consider a function that builds a pizza. It needs to accept a number of toppings, but you can't know ahead of time how many toppings a person will want. The function in the following example has one parameter, `*toppings`, but this parameter collects as many arguments as the calling line provides:

```
pizza.py    def make_pizza(*toppings):
                """Print the list of toppings that have been requested."""
                print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

The asterisk in the parameter name `*toppings` tells Python to make an empty tuple called `toppings` and pack whatever values it receives into this tuple. The `print()` call in the function body produces output showing that Python can handle a function call with one value and a call with three values. It treats the different calls similarly. Note that Python packs the arguments into a tuple, even if the function receives only one value:

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

Now we can replace the `print()` call with a loop that runs through the list of toppings and describes the pizza being ordered:

```
def make_pizza(*toppings):
    """Summarize the pizza we are about to make."""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

The function responds appropriately, whether it receives one value or three values:

```
Making a pizza with the following toppings:
- pepperoni
```

```
Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

This syntax works no matter how many arguments the function receives.

Mixing Positional and Arbitrary Arguments

If you want a function to accept several different kinds of arguments, the parameter that accepts an arbitrary number of arguments must be placed last in the function definition. Python matches positional and keyword arguments first and then collects any remaining arguments in the final parameter.

For example, if the function needs to take in a size for the pizza, that parameter must come before the parameter `*toppings`:

```
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

In the function definition, Python assigns the first value it receives to the parameter `size`. All other values that come after are stored in the tuple `toppings`. The function calls include an argument for the size first, followed by as many toppings as needed.

Now each pizza has a size and a number of toppings, and each piece of information is printed in the proper place, showing size first and toppings after:

```
Making a 16-inch pizza with the following toppings:
```

```
- pepperoni
```

```
Making a 12-inch pizza with the following toppings:
```

```
- mushrooms
- green peppers
- extra cheese
```

NOTE

You'll often see the generic parameter name `*args`, which collects arbitrary positional arguments like this.

Using Arbitrary Keyword Arguments

Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. In this case, you can write functions that accept as many key-value pairs as the calling statement provides. One example involves building user profiles: you know you'll get information about a user, but you're not sure what kind of information you'll receive. The function `build_profile()` in the

following example always takes in a first and last name, but it accepts an arbitrary number of keyword arguments as well:

```
user_profile.py def build_profile(first, last, **user_info):
    """Build a dictionary containing everything we know about a user."""
❶    user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info

user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='physics')
print(user_profile)
```

The definition of `build_profile()` expects a first and last name, and then it allows the user to pass in as many name-value pairs as they want. The double asterisks before the parameter `**user_info` cause Python to create an empty dictionary called `user_info` and pack whatever name-value pairs it receives into this dictionary. Within the function, you can access the key-value pairs in `user_info` just as you would for any dictionary.

In the body of `build_profile()`, we add the first and last names to the `user_info` dictionary because we'll always receive these two pieces of information from the user ❶, and they haven't been placed into the dictionary yet. Then we return the `user_info` dictionary to the function call line.

We call `build_profile()`, passing it the first name '`albert`', the last name '`einstein`', and the two key-value pairs `location='princeton'` and `field='physics'`. We assign the returned profile to `user_profile` and print `user_profile`:

```
{'location': 'princeton', 'field': 'physics',
'first_name': 'albert', 'last_name': 'einstein'}
```

The returned dictionary contains the user's first and last names and, in this case, the location and field of study as well. The function would work no matter how many additional key-value pairs are provided in the function call.

You can mix positional, keyword, and arbitrary values in many different ways when writing your own functions. It's useful to know that all these argument types exist because you'll see them often when you start reading other people's code. It takes practice to learn to use the different types correctly and to know when to use each type. For now, remember to use the simplest approach that gets the job done. As you progress you'll learn to use the most efficient approach each time.

NOTE

*You'll often see the parameter name `**kwargs` used to collect non-specific keyword arguments.*

TRY IT YOURSELF

8-12. Sandwiches: Write a function that accepts a list of items a person wants on a sandwich. The function should have one parameter that collects as many items as the function call provides, and it should print a summary of the sandwich that's being ordered. Call the function three times, using a different number of arguments each time.

8-13. User Profile: Start with a copy of `user_profile.py` from page 149. Build a profile of yourself by calling `build_profile()`, using your first and last names and three other key-value pairs that describe you.

8-14. Cars: Write a function that stores information about a car in a dictionary. The function should always receive a manufacturer and a model name. It should then accept an arbitrary number of keyword arguments. Call the function with the required information and two other name-value pairs, such as a color or an optional feature. Your function should work for a call like this one:

```
car = make_car('subaru', 'outback', color='blue', tow_package=True)
```

Print the dictionary that's returned to make sure all the information was stored correctly.

Storing Your Functions in Modules

One advantage of functions is the way they separate blocks of code from your main program. By using descriptive names for your functions, your main program will be much easier to follow. You can go a step further by storing your functions in a separate file called a *module* and then *importing* that module into your main program. An `import` statement tells Python to make the code in a module available in the currently running program file.

Storing your functions in a separate file allows you to hide the details of your program's code and focus on its higher-level logic. It also allows you to reuse functions in many different programs. When you store your functions in separate files, you can share those files with other programmers without having to share your entire program. Knowing how to import functions also allows you to use libraries of functions that other programmers have written.

There are several ways to import a module, and I'll show you each of these briefly.

Importing an Entire Module

To start importing functions, we first need to create a module. A *module* is a file ending in `.py` that contains the code you want to import into your

program. Let's make a module that contains the function `make_pizza()`. To make this module, we'll remove everything from the file `pizza.py` except the function `make_pizza()`:

```
pizza.py
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
```

Now we'll make a separate file called `making_pizzas.py` in the same directory as `pizza.py`. This file imports the module we just created and then makes two calls to `make_pizza()`:

```
making
_pizzas.py
import pizza
❶ pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

When Python reads this file, the line `import pizza` tells Python to open the file `pizza.py` and copy all the functions from it into this program. You don't actually see code being copied between files because Python copies the code behind the scenes just before the program runs. All you need to know is that any function defined in `pizza.py` will now be available in `making_pizzas.py`.

To call a function from an imported module, enter the name of the module you imported, `pizza`, followed by the name of the function, `make_pizza()`, separated by a dot ❶. This code produces the same output as the original program that didn't import a module:

```
Making a 16-inch pizza with the following toppings:
- pepperoni
```

```
Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

This first approach to importing, in which you simply write `import` followed by the name of the module, makes every function from the module available in your program. If you use this kind of `import` statement to import an entire module named `module_name.py`, each function in the module is available through the following syntax:

```
module_name.function_name()
```

Importing Specific Functions

You can also import a specific function from a module. Here's the general syntax for this approach:

```
from module_name import function_name
```

You can import as many functions as you want from a module by separating each function's name with a comma:

```
from module_name import function_0, function_1, function_2
```

The *making_pizzas.py* example would look like this if we want to import just the function we're going to use:

```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

With this syntax, you don't need to use the dot notation when you call a function. Because we've explicitly imported the function `make_pizza()` in the `import` statement, we can call it by name when we use the function.

Using `as` to Give a Function an Alias

If the name of a function you're importing might conflict with an existing name in your program or if the function name is long, you can use a short, unique *alias*—an alternate name similar to a nickname for the function. You'll give the function this special nickname when you import the function.

Here we give the function `make_pizza()` an alias, `mp()`, by importing `make_pizza` as `mp`. The `as` keyword renames a function using the alias you provide:

```
from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

The `import` statement shown here renames the function `make_pizza()` to `mp()` in this program. Any time we want to call `make_pizza()` we can simply write `mp()` instead, and Python will run the code in `make_pizza()` while avoiding any confusion with another `make_pizza()` function you might have written in this program file.

The general syntax for providing an alias is:

```
from module_name import function_name as fn
```

Using `as` to Give a Module an Alias

You can also provide an alias for a module name. Giving a module a short alias, like `p` for `pizza`, allows you to call the module's functions more quickly. Calling `p.make_pizza()` is more concise than calling `pizza.make_pizza()`:

```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

The module `pizza` is given the alias `p` in the `import` statement, but all of the module's functions retain their original names. Calling the functions by writing `p.make_pizza()` is not only more concise than writing `pizza.make_pizza()`, but also redirects your attention from the module name and allows you to focus on the descriptive names of its functions. These function names, which clearly tell you what each function does, are more important to the readability of your code than using the full module name.

The general syntax for this approach is:

```
import module_name as mn
```

Importing All Functions in a Module

You can tell Python to import every function in a module by using the asterisk (*) operator:

```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

The asterisk in the `import` statement tells Python to copy every function from the module `pizza` into this program file. Because every function is imported, you can call each function by name without using the dot notation. However, it's best not to use this approach when you're working with larger modules that you didn't write: if the module has a function name that matches an existing name in your project, you can get some unexpected results. Python may see several functions or variables with the same name, and instead of importing all the functions separately, it will overwrite the functions.

The best approach is to import the function or functions you want, or import the entire module and use the dot notation. This leads to clear code that's easy to read and understand. I include this section so you'll recognize `import` statements like the following when you see them in other people's code:

```
from module_name import *
```

Styling Functions

You need to keep a few details in mind when you’re styling functions. Functions should have descriptive names, and these names should use lowercase letters and underscores. Descriptive names help you and others understand what your code is trying to do. Module names should use these conventions as well.

Every function should have a comment that explains concisely what the function does. This comment should appear immediately after the function definition and use the docstring format. In a well-documented function, other programmers can use the function by reading only the description in the docstring. They should be able to trust that the code works as described, and as long as they know the name of the function, the arguments it needs, and the kind of value it returns, they should be able to use it in their programs.

If you specify a default value for a parameter, no spaces should be used on either side of the equal sign:

```
def function_name(parameter_0, parameter_1='default value')
```

The same convention should be used for keyword arguments in function calls:

```
function_name(value_0, parameter_1='value')
```

PEP 8 (<https://www.python.org/dev/peps/pep-0008/>) recommends that you limit lines of code to 79 characters so every line is visible in a reasonably sized editor window. If a set of parameters causes a function’s definition to be longer than 79 characters, press ENTER after the opening parenthesis on the definition line. On the next line, press TAB twice to separate the list of arguments from the body of the function, which will only be indented one level.

Most editors automatically line up any additional lines of parameters to match the indentation you have established on the first line:

```
def function_name(
    parameter_0, parameter_1, parameter_2,
    parameter_3, parameter_4, parameter_5):
    function body...
```

If your program or module has more than one function, you can separate each by two blank lines to make it easier to see where one function ends and the next one begins.

All `import` statements should be written at the beginning of a file. The only exception is if you use comments at the beginning of your file to describe the overall program.

TRY IT YOURSELF

8-15. Printing Models: Put the functions for the example *printing_models.py* in a separate file called *printing_functions.py*. Write an `import` statement at the top of *printing_models.py*, and modify the file to use the imported functions.

8-16. Imports: Using a program you wrote that has one function in it, store that function in a separate file. Import the function into your main program file, and call the function using each of these approaches:

```
import module_name
from module_name import function_name
from module_name import function_name as fn
import module_name as mn
from module_name import *
```

8-17. Styling Functions: Choose any three programs you wrote for this chapter, and make sure they follow the styling guidelines described in this section.

Summary

In this chapter you learned how to write functions and to pass arguments so that your functions have access to the information they need to do their work. You learned how to use positional and keyword arguments, and how to accept an arbitrary number of arguments. You saw functions that display output and functions that return values. You learned how to use functions with lists, dictionaries, if statements, and while loops. You also saw how to store your functions in separate files called *modules*, so your program files will be simpler and easier to understand. Finally, you learned to style your functions so your programs will continue to be well-structured and as easy as possible for you and others to read.

One of your goals as a programmer should be to write simple code that does what you want it to, and functions help you do this. They allow you to write blocks of code and leave them alone once you know they work. When you know a function does its job correctly, you can trust that it will continue to work and move on to your next coding task.

Functions allow you to write code once and then reuse that code as many times as you want. When you need to run the code in a function, all you need to do is write a one-line call and the function does its job. When you need to modify a function's behavior, you only have to modify one block of code, and your change takes effect everywhere you've made a call to that function.

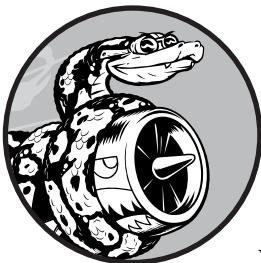
Using functions makes your programs easier to read, and good function names summarize what each part of a program does. Reading a series of function calls gives you a much quicker sense of what a program does than reading a long series of code blocks.

Functions also make your code easier to test and debug. When the bulk of your program’s work is done by a set of functions, each of which has a specific job, it’s much easier to test and maintain the code you’ve written. You can write a separate program that calls each function and tests whether each function works in all the situations it may encounter. When you do this, you can be confident that your functions will work properly each time you call them.

In Chapter 9 you’ll learn to write classes. *Classes* combine functions and data into one neat package that can be used in flexible and efficient ways.

9

CLASSES



Object-oriented programming is one of the most effective approaches to writing software. In object-oriented programming you write *classes* that represent real-world things and situations, and you create *objects* based on these classes. When you write a class, you define the general behavior that a whole category of objects can have.

When you create individual objects from the class, each object is automatically equipped with the general behavior; you can then give each object whatever unique traits you desire. You'll be amazed how well real-world situations can be modeled with object-oriented programming.

Making an object from a class is called *instantiation*, and you work with *instances* of a class. In this chapter you'll write classes and create instances of those classes. You'll specify the kind of information that can be stored in instances, and you'll define actions that can be taken with these instances. You'll also write classes that extend the functionality of existing classes, so

similar classes can share code efficiently. You'll store your classes in modules and import classes written by other programmers into your own program files.

Understanding object-oriented programming will help you see the world as a programmer does. It'll help you really know your code, not just what's happening line by line, but also the bigger concepts behind it. Knowing the logic behind classes will train you to think logically so you can write programs that effectively address almost any problem you encounter.

Classes also make life easier for you and the other programmers you'll work with as you take on increasingly complex challenges. When you and other programmers write code based on the same kind of logic, you'll be able to understand each other's work. Your programs will make sense to many collaborators, allowing everyone to accomplish more.

Creating and Using a Class

You can model almost anything using classes. Let's start by writing a simple class, `Dog`, that represents a dog—not one dog in particular, but any dog. What do we know about most pet dogs? Well, they all have a name and age. We also know that most dogs sit and roll over. Those two pieces of information (name and age) and those two behaviors (sit and roll over) will go in our `Dog` class because they're common to most dogs. This class will tell Python how to make an object representing a dog. After our class is written, we'll use it to make individual instances, each of which represents one specific dog.

Creating the Dog Class

Each instance created from the `Dog` class will store a `name` and an `age`, and we'll give each dog the ability to `sit()` and `roll_over()`:

```
dog.py ❶ class Dog:
❷     """A simple attempt to model a dog."""

❸     def __init__(self, name, age):
❹         """Initialize name and age attributes."""
❺         self.name = name
❻         self.age = age

❼     def sit(self):
⪻         """Simulate a dog sitting in response to a command."""
⪼         print(f"{self.name} is now sitting.")

⪽     def roll_over(self):
⪾         """Simulate rolling over in response to a command."""
⪿         print(f"{self.name} rolled over!")
```

There's a lot to notice here, but don't worry. You'll see this structure throughout this chapter and have lots of time to get used to it. At ❶ we

define a class called `Dog`. By convention, capitalized names refer to classes in Python. There are no parentheses in the class definition because we’re creating this class from scratch. At ❷ we write a docstring describing what this class does.

The `__init__()` Method

A function that’s part of a class is a *method*. Everything you learned about functions applies to methods as well; the only practical difference for now is the way we’ll call methods. The `__init__()` method at ❸ is a special method that Python runs automatically whenever we create a new instance based on the `Dog` class. This method has two leading underscores and two trailing underscores, a convention that helps prevent Python’s default method names from conflicting with your method names. Make sure to use two underscores on each side of `__init__()`. If you use just one on each side, the method won’t be called automatically when you use your class, which can result in errors that are difficult to identify.

We define the `__init__()` method to have three parameters: `self`, `name`, and `age`. The `self` parameter is required in the method definition, and it must come first before the other parameters. It must be included in the definition because when Python calls this method later (to create an instance of `Dog`), the method call will automatically pass the `self` argument. Every method call associated with an instance automatically passes `self`, which is a reference to the instance itself; it gives the individual instance access to the attributes and methods in the class. When we make an instance of `Dog`, Python will call the `__init__()` method from the `Dog` class. We’ll pass `Dog()` a name and an age as arguments; `self` is passed automatically, so we don’t need to pass it. Whenever we want to make an instance from the `Dog` class, we’ll provide values for only the last two parameters, `name` and `age`.

The two variables defined at ❹ each have the prefix `self`. Any variable prefixed with `self` is available to every method in the class, and we’ll also be able to access these variables through any instance created from the class. The line `self.name = name` takes the value associated with the parameter `name` and assigns it to the variable `name`, which is then attached to the instance being created. The same process happens with `self.age = age`. Variables that are accessible through instances like this are called *attributes*.

The `Dog` class has two other methods defined: `sit()` and `roll_over()` ❺. Because these methods don’t need additional information to run, we just define them to have one parameter, `self`. The instances we create later will have access to these methods. In other words, they’ll be able to sit and roll over. For now, `sit()` and `roll_over()` don’t do much. They simply print a message saying the dog is sitting or rolling over. But the concept can be extended to realistic situations: if this class were part of an actual computer game, these methods would contain code to make an animated dog sit and roll over. If this class was written to control a robot, these methods would direct movements that cause a robotic dog to sit and roll over.

Making an Instance from a Class

Think of a class as a set of instructions for how to make an instance. The class `Dog` is a set of instructions that tells Python how to make individual instances representing specific dogs.

Let's make an instance representing a specific dog:

```
class Dog:  
    --snip--  
  
❶ my_dog = Dog('Willie', 6)  
  
❷ print(f"My dog's name is {my_dog.name}.")  
❸ print(f"My dog is {my_dog.age} years old.")
```

The `Dog` class we're using here is the one we just wrote in the previous example. At ❶ we tell Python to create a dog whose name is 'Willie' and whose age is 6. When Python reads this line, it calls the `__init__()` method in `Dog` with the arguments 'Willie' and 6. The `__init__()` method creates an instance representing this particular dog and sets the `name` and `age` attributes using the values we provided. Python then returns an instance representing this dog. We assign that instance to the variable `my_dog`. The naming convention is helpful here: we can usually assume that a capitalized name like `Dog` refers to a class, and a lowercase name like `my_dog` refers to a single instance created from a class.

Accessing Attributes

To access the attributes of an instance, you use dot notation. At ❷ we access the value of `my_dog`'s attribute `name` by writing:

```
my_dog.name
```

Dot notation is used often in Python. This syntax demonstrates how Python finds an attribute's value. Here Python looks at the instance `my_dog` and then finds the attribute `name` associated with `my_dog`. This is the same attribute referred to as `self.name` in the class `Dog`. At ❸ we use the same approach to work with the attribute `age`.

The output is a summary of what we know about `my_dog`:

```
My dog's name is Willie.  
My dog is 6 years old.
```

Calling Methods

After we create an instance from the class `Dog`, we can use dot notation to call any method defined in `Dog`. Let's make our dog sit and roll over:

```
class Dog:  
    --snip--
```

```
my_dog = Dog('Willie', 6)
my_dog.sit()
my_dog.roll_over()
```

To call a method, give the name of the instance (in this case, `my_dog`) and the method you want to call, separated by a dot. When Python reads `my_dog.sit()`, it looks for the method `sit()` in the class `Dog` and runs that code. Python interprets the line `my_dog.roll_over()` in the same way.

Now Willie does what we tell him to:

```
Willie is now sitting.
Willie rolled over!
```

This syntax is quite useful. When attributes and methods have been given appropriately descriptive names like `name`, `age`, `sit()`, and `roll_over()`, we can easily infer what a block of code, even one we've never seen before, is supposed to do.

Creating Multiple Instances

You can create as many instances from a class as you need. Let's create a second dog called `your_dog`:

```
class Dog:
    --snip--

    my_dog = Dog('Willie', 6)
    your_dog = Dog('Lucy', 3)

    print(f"My dog's name is {my_dog.name}.")
    print(f"My dog is {my_dog.age} years old.")
    my_dog.sit()

    print(f"\nYour dog's name is {your_dog.name}.")
    print(f"Your dog is {your_dog.age} years old.")
    your_dog.sit()
```

In this example we create a dog named Willie and a dog named Lucy. Each dog is a separate instance with its own set of attributes, capable of the same set of actions:

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

Even if we used the same name and age for the second dog, Python would still create a separate instance from the `Dog` class. You can make

as many instances from one class as you need, as long as you give each instance a unique variable name or it occupies a unique spot in a list or dictionary.

TRY IT YOURSELF

9-1. Restaurant: Make a class called Restaurant. The `__init__()` method for Restaurant should store two attributes: a `restaurant_name` and a `cuisine_type`. Make a method called `describe_restaurant()` that prints these two pieces of information, and a method called `open_restaurant()` that prints a message indicating that the restaurant is open.

Make an instance called `restaurant` from your class. Print the two attributes individually, and then call both methods.

9-2. Three Restaurants: Start with your class from Exercise 9-1. Create three different instances from the class, and call `describe_restaurant()` for each instance.

9-3. Users: Make a class called User. Create two attributes called `first_name` and `last_name`, and then create several other attributes that are typically stored in a user profile. Make a method called `describe_user()` that prints a summary of the user's information. Make another method called `greet_user()` that prints a personalized greeting to the user.

Create several instances representing different users, and call both methods for each user.

Working with Classes and Instances

You can use classes to represent many real-world situations. Once you write a class, you'll spend most of your time working with instances created from that class. One of the first tasks you'll want to do is modify the attributes associated with a particular instance. You can modify the attributes of an instance directly or write methods that update attributes in specific ways.

The Car Class

Let's write a new class representing a car. Our class will store information about the kind of car we're working with, and it will have a method that summarizes this information:

```
car.py    class Car:  
              """A simple attempt to represent a car."""  
  
        ①     def __init__(self, make, model, year):  
                  """Initialize attributes to describe a car."""  
                  self.make = make
```

```
    self.model = model
    self.year = year

❷ def get_descriptive_name(self):
    """Return a neatly formatted descriptive name."""
    long_name = f'{self.year} {self.manufacturer} {self.model}'
    return long_name.title()

❸ my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
```

At ❶ in the `Car` class, we define the `__init__()` method with the `self` parameter first, just like we did before with our `Dog` class. We also give it three other parameters: `make`, `model`, and `year`. The `__init__()` method takes in these parameters and assigns them to the attributes that will be associated with instances made from this class. When we make a new `Car` instance, we'll need to specify a `make`, `model`, and `year` for our instance.

At ❷ we define a method called `get_descriptive_name()` that puts a car's `year`, `make`, and `model` into one string neatly describing the car. This will spare us from having to print each attribute's value individually. To work with the attribute values in this method, we use `self.make`, `self.model`, and `self.year`. At ❸ we make an instance from the `Car` class and assign it to the variable `my_new_car`. Then we call `get_descriptive_name()` to show what kind of car we have:

```
2019 Audi A4
```

To make the class more interesting, let's add an attribute that changes over time. We'll add an attribute that stores the car's overall mileage.

Setting a Default Value for an Attribute

When an instance is created, attributes can be defined without being passed in as parameters. These attributes can be defined in the `__init__()` method, where they are assigned a default value.

Let's add an attribute called `odometer_reading` that always starts with a value of 0. We'll also add a method `read_odometer()` that helps us read each car's odometer:

```
class Car:

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
❶        self.odometer_reading = 0

    def get_descriptive_name(self):
        --snip--
```

```
❷     def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

This time when Python calls the `_init_()` method to create a new instance, it stores the make, model, and year values as attributes like it did in the previous example. Then Python creates a new attribute called `odometer_reading` and sets its initial value to 0 ❶. We also have a new method called `read_odometer()` at ❷ that makes it easy to read a car's mileage.

Our car starts with a mileage of 0:

```
2019 Audi A4
This car has 0 miles on it.
```

Not many cars are sold with exactly 0 miles on the odometer, so we need a way to change the value of this attribute.

Modifying Attribute Values

You can change an attribute's value in three ways: you can change the value directly through an instance, set the value through a method, or increment the value (add a certain amount to it) through a method. Let's look at each of these approaches.

Modifying an Attribute's Value Directly

The simplest way to modify the value of an attribute is to access the attribute directly through an instance. Here we set the odometer reading to 23 directly:

```
class Car:
    --snip--

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

❶ my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

At ❶ we use dot notation to access the car's `odometer_reading` attribute and set its value directly. This line tells Python to take the instance `my_new_car`, find the attribute `odometer_reading` associated with it, and set the value of that attribute to 23:

```
2019 Audi A4
This car has 23 miles on it.
```

Sometimes you'll want to access attributes directly like this, but other times you'll want to write a method that updates the value for you.

Modifying an Attribute's Value Through a Method

It can be helpful to have methods that update certain attributes for you. Instead of accessing the attribute directly, you pass the new value to a method that handles the updating internally.

Here's an example showing a method called `update_odometer()`:

```
class Car:  
    --snip--  
  
❶    def update_odometer(self, mileage):  
        """Set the odometer reading to the given value."""  
        self.odometer_reading = mileage  
  
my_new_car = Car('audi', 'a4', 2019)  
print(my_new_car.get_descriptive_name())  
  
❷ my_new_car.update_odometer(23)  
my_new_car.read_odometer()
```

The only modification to `Car` is the addition of `update_odometer()` at ❶. This method takes in a mileage value and assigns it to `self.odometer_reading`. At ❷ we call `update_odometer()` and give it 23 as an argument (corresponding to the `mileage` parameter in the method definition). It sets the odometer reading to 23, and `read_odometer()` prints the reading:

```
2019 Audi A4  
This car has 23 miles on it.
```

We can extend the method `update_odometer()` to do additional work every time the odometer reading is modified. Let's add a little logic to make sure no one tries to roll back the odometer reading:

```
class Car:  
    --snip--  
  
    def update_odometer(self, mileage):  
        """  
        Set the odometer reading to the given value.  
        Reject the change if it attempts to roll the odometer back.  
        """  
  
❶        if mileage >= self.odometer_reading:  
            self.odometer_reading = mileage  
        else:  
❷            print("You can't roll back an odometer!")
```

Now `update_odometer()` checks that the new reading makes sense before modifying the attribute. If the new mileage, `mileage`, is greater than or equal

to the existing mileage, `self.odometer_reading`, you can update the odometer reading to the new mileage ❶. If the new mileage is less than the existing mileage, you'll get a warning that you can't roll back an odometer ❷.

Incrementing an Attribute's Value Through a Method

Sometimes you'll want to increment an attribute's value by a certain amount rather than set an entirely new value. Say we buy a used car and put 100 miles on it between the time we buy it and the time we register it. Here's a method that allows us to pass this incremental amount and add that value to the odometer reading:

```
class Car:  
    --snip--  
  
    def update_odometer(self, mileage):  
        --snip--  
  
❶    def increment_odometer(self, miles):  
        """Add the given amount to the odometer reading."""  
        self.odometer_reading += miles  
  
❷    my_used_car = Car('subaru', 'outback', 2015)  
    print(my_used_car.get_descriptive_name())  
  
❸    my_used_car.update_odometer(23_500)  
    my_used_car.read_odometer()  
  
❹    my_used_car.increment_odometer(100)  
    my_used_car.read_odometer()
```

The new method `increment_odometer()` at ❶ takes in a number of miles, and adds this value to `self.odometer_reading`. At ❷ we create a used car, `my_used_car`. We set its odometer to 23,500 by calling `update_odometer()` and passing it 23_500 at ❸. At ❹ we call `increment_odometer()` and pass it 100 to add the 100 miles that we drove between buying the car and registering it:

```
2015 Subaru Outback  
This car has 23500 miles on it.  
This car has 23600 miles on it.
```

You can easily modify this method to reject negative increments so no one uses this function to roll back an odometer.

NOTE

You can use methods like this to control how users of your program update values such as an odometer reading, but anyone with access to the program can set the odometer reading to any value by accessing the attribute directly. Effective security takes extreme attention to detail in addition to basic checks like those shown here.

TRY IT YOURSELF

9-4. Number Served: Start with your program from Exercise 9-1 (page 162).

Add an attribute called `number_served` with a default value of 0. Create an instance called `restaurant` from this class. Print the number of customers the restaurant has served, and then change this value and print it again.

Add a method called `set_number_served()` that lets you set the number of customers that have been served. Call this method with a new number and print the value again.

Add a method called `increment_number_served()` that lets you increment the number of customers who've been served. Call this method with any number you like that could represent how many customers were served in, say, a day of business.

9-5. Login Attempts: Add an attribute called `login_attempts` to your `User` class from Exercise 9-3 (page 162). Write a method called `increment_login_attempts()` that increments the value of `login_attempts` by 1. Write another method called `reset_login_attempts()` that resets the value of `login_attempts` to 0.

Make an instance of the `User` class and call `increment_login_attempts()` several times. Print the value of `login_attempts` to make sure it was incremented properly, and then call `reset_login_attempts()`. Print `login_attempts` again to make sure it was reset to 0.

Inheritance

You don't always have to start from scratch when writing a class. If the class you're writing is a specialized version of another class you wrote, you can use *inheritance*. When one class *inherits* from another, it takes on the attributes and methods of the first class. The original class is called the *parent class*, and the new class is the *child class*. The child class can inherit any or all of the attributes and methods of its parent class, but it's also free to define new attributes and methods of its own.

The `__init__()` Method for a Child Class

When you're writing a new class based on an existing class, you'll often want to call the `__init__()` method from the parent class. This will initialize any attributes that were defined in the parent `__init__()` method and make them available in the child class.

As an example, let's model an electric car. An electric car is just a specific kind of car, so we can base our new `ElectricCar` class on the `Car` class we wrote earlier. Then we'll only have to write code for the attributes and behavior specific to electric cars.

Let's start by making a simple version of the ElectricCar class, which does everything the Car class does:

```
electric_car.py ❶ class Car:  
    """A simple attempt to represent a car."""  
  
    def __init__(self, make, model, year):  
        self.make = make  
        self.model = model  
        self.year = year  
        self.odometer_reading = 0  
  
    def get_descriptive_name(self):  
        long_name = f"{self.year} {self.manufacturer} {self.model}"  
        return long_name.title()  
  
    def read_odometer(self):  
        print(f"This car has {self.odometer_reading} miles on it.")  
  
    def update_odometer(self, mileage):  
        if mileage >= self.odometer_reading:  
            self.odometer_reading = mileage  
        else:  
            print("You can't roll back an odometer!")  
  
    def increment_odometer(self, miles):  
        self.odometer_reading += miles  
  
❷ class ElectricCar(Car):  
    """Represent aspects of a car, specific to electric vehicles."""  
  
❸     def __init__(self, make, model, year):  
        """Initialize attributes of the parent class."""  
❹     super().__init__(make, model, year)  
  
❺ my_tesla = ElectricCar('tesla', 'model s', 2019)  
print(my_tesla.get_descriptive_name())
```

At ❶ we start with Car. When you create a child class, the parent class must be part of the current file and must appear before the child class in the file. At ❷ we define the child class, ElectricCar. The name of the parent class must be included in parentheses in the definition of a child class. The `__init__()` method at ❸ takes in the information required to make a Car instance.

The `super()` function at ❹ is a special function that allows you to call a method from the parent class. This line tells Python to call the `__init__()` method from Car, which gives an ElectricCar instance all the attributes defined in that method. The name `super` comes from a convention of calling the parent class a *superclass* and the child class a *subclass*.

We test whether inheritance is working properly by trying to create an electric car with the same kind of information we'd provide when making a regular car. At ❸ we make an instance of the `ElectricCar` class and assign it to `my_tesla`. This line calls the `__init__()` method defined in `ElectricCar`, which in turn tells Python to call the `__init__()` method defined in the parent class `Car`. We provide the arguments '`tesla`', '`model s`', and `2019`.

Aside from `__init__()`, there are no attributes or methods yet that are particular to an electric car. At this point we're just making sure the electric car has the appropriate `Car` behaviors:

2019 Tesla Model S

The `ElectricCar` instance works just like an instance of `Car`, so now we can begin defining attributes and methods specific to electric cars.

Defining Attributes and Methods for the Child Class

Once you have a child class that inherits from a parent class, you can add any new attributes and methods necessary to differentiate the child class from the parent class.

Let's add an attribute that's specific to electric cars (a battery, for example) and a method to report on this attribute. We'll store the battery size and write a method that prints a description of the battery:

```
class Car:  
    --snip--  
  
class ElectricCar(Car):  
    """Represent aspects of a car, specific to electric vehicles."""  
  
    def __init__(self, make, model, year):  
        """  
        Initialize attributes of the parent class.  
        Then initialize attributes specific to an electric car.  
        """  
        super().__init__(make, model, year)  
        self.battery_size = 75  
  
    def describe_battery(self):  
        """Print a statement describing the battery size."""  
        print(f"This car has a {self.battery_size}-kWh battery.)  
  
my_tesla = ElectricCar('tesla', 'model s', 2019)  
print(my_tesla.get_descriptive_name())  
my_tesla.describe_battery()
```

At ❶ we add a new attribute `self.battery_size` and set its initial value to, say, 75. This attribute will be associated with all instances created from the `ElectricCar` class but won't be associated with any instances of `Car`. We also

add a method called `describe_battery()` that prints information about the battery at ❷. When we call this method, we get a description that is clearly specific to an electric car:

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

There's no limit to how much you can specialize the `ElectricCar` class. You can add as many attributes and methods as you need to model an electric car to whatever degree of accuracy you need. An attribute or method that could belong to any car, rather than one that's specific to an electric car, should be added to the `Car` class instead of the `ElectricCar` class. Then anyone who uses the `Car` class will have that functionality available as well, and the `ElectricCar` class will only contain code for the information and behavior specific to electric vehicles.

Overriding Methods from the Parent Class

You can override any method from the parent class that doesn't fit what you're trying to model with the child class. To do this, you define a method in the child class with the same name as the method you want to override in the parent class. Python will disregard the parent class method and only pay attention to the method you define in the child class.

Say the class `Car` had a method called `fill_gas_tank()`. This method is meaningless for an all-electric vehicle, so you might want to override this method. Here's one way to do that:

```
class ElectricCar(Car):
    --snip--

    def fill_gas_tank(self):
        """Electric cars don't have gas tanks."""
        print("This car doesn't need a gas tank!")
```

Now if someone tries to call `fill_gas_tank()` with an electric car, Python will ignore the method `fill_gas_tank()` in `Car` and run this code instead. When you use inheritance, you can make your child classes retain what you need and override anything you don't need from the parent class.

Instances as Attributes

When modeling something from the real world in code, you may find that you're adding more and more detail to a class. You'll find that you have a growing list of attributes and methods and that your files are becoming lengthy. In these situations, you might recognize that part of one class can be written as a separate class. You can break your large class into smaller classes that work together.

For example, if we continue adding detail to the `ElectricCar` class, we might notice that we're adding many attributes and methods specific to

the car's battery. When we see this happening, we can stop and move those attributes and methods to a separate class called `Battery`. Then we can use a `Battery` instance as an attribute in the `ElectricCar` class:

```
class Car:  
    --snip--  
  
❶ class Battery:  
    """A simple attempt to model a battery for an electric car."""  
  
❷     def __init__(self, battery_size=75):  
        """Initialize the battery's attributes."""  
        self.battery_size = battery_size  
  
❸     def describe_battery(self):  
        """Print a statement describing the battery size."""  
        print(f"This car has a {self.battery_size}-kWh battery.")  
  
class ElectricCar(Car):  
    """Represent aspects of a car, specific to electric vehicles."""  
  
    def __init__(self, make, model, year):  
        """Initialize attributes of the parent class.  
        Then initialize attributes specific to an electric car.  
        """  
        super().__init__(make, model, year)  
❹     self.battery = Battery()  
  
my_tesla = ElectricCar('tesla', 'model s', 2019)  
  
print(my_tesla.get_descriptive_name())  
my_tesla.battery.describe_battery()
```

At ❶ we define a new class called `Battery` that doesn't inherit from any other class. The `__init__()` method at ❷ has one parameter, `battery_size`, in addition to `self`. This is an optional parameter that sets the battery's size to 75 if no value is provided. The method `describe_battery()` has been moved to this class as well ❸.

In the `ElectricCar` class, we now add an attribute called `self.battery` ❹. This line tells Python to create a new instance of `Battery` (with a default size of 75, because we're not specifying a value) and assign that instance to the attribute `self.battery`. This will happen every time the `__init__()` method is called; any `ElectricCar` instance will now have a `Battery` instance created automatically.

We create an electric car and assign it to the variable `my_tesla`. When we want to describe the battery, we need to work through the car's battery attribute:

```
my_tesla.battery.describe_battery()
```

This line tells Python to look at the instance `my_tesla`, find its `battery` attribute, and call the method `describe_battery()` that's associated with the `Battery` instance stored in the attribute.

The output is identical to what we saw previously:

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

This looks like a lot of extra work, but now we can describe the battery in as much detail as we want without cluttering the `ElectricCar` class. Let's add another method to `Battery` that reports the range of the car based on the battery size:

```
class Car:
    --snip--

class Battery:
    --snip--

❶ def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315

        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    --snip--

    my_tesla = ElectricCar('tesla', 'model s', 2019)
    print(my_tesla.get_descriptive_name())
    my_tesla.battery.describe_battery()
❷ my_tesla.battery.get_range()
```

The new method `get_range()` at ❶ performs some simple analysis. If the battery's capacity is 75 kWh, `get_range()` sets the range to 260 miles, and if the capacity is 100 kWh, it sets the range to 315 miles. It then reports this value. When we want to use this method, we again have to call it through the car's `battery` attribute at ❷.

The output tells us the range of the car based on its battery size:

```
2019 Tesla Model S
This car has a 75-kWh battery.
This car can go about 260 miles on a full charge.
```

Modeling Real-World Objects

As you begin to model more complicated things like electric cars, you'll wrestle with interesting questions. Is the range of an electric car a property of the battery or of the car? If we're only describing one car, it's probably fine to maintain the association of the method `get_range()` with the `Battery` class. But if we're describing a manufacturer's entire line of cars, we probably want to move `get_range()` to the `ElectricCar` class. The `get_range()` method would still check the battery size before determining the range, but it would report a range specific to the kind of car it's associated with. Alternatively, we could maintain the association of the `get_range()` method with the battery but pass it a parameter such as `car_model`. The `get_range()` method would then report a range based on the battery size and car model.

This brings you to an interesting point in your growth as a programmer. When you wrestle with questions like these, you're thinking at a higher logical level rather than a syntax-focused level. You're thinking not about Python, but about how to represent the real world in code. When you reach this point, you'll realize there are often no right or wrong approaches to modeling real-world situations. Some approaches are more efficient than others, but it takes practice to find the most efficient representations. If your code is working as you want it to, you're doing well! Don't be discouraged if you find you're ripping apart your classes and rewriting them several times using different approaches. In the quest to write accurate, efficient code, everyone goes through this process.

TRY IT YOURSELF

9-6. Ice Cream Stand: An ice cream stand is a specific kind of restaurant. Write a class called `IceCreamStand` that inherits from the `Restaurant` class you wrote in Exercise 9-1 (page 162) or Exercise 9-4 (page 167). Either version of the class will work; just pick the one you like better. Add an attribute called `flavors` that stores a list of ice cream flavors. Write a method that displays these flavors. Create an instance of `IceCreamStand`, and call this method.

9-7. Admin: An administrator is a special kind of user. Write a class called `Admin` that inherits from the `User` class you wrote in Exercise 9-3 (page 162) or Exercise 9-5 (page 167). Add an attribute, `privileges`, that stores a list of strings like "can add post", "can delete post", "can ban user", and so on. Write a method called `show_privileges()` that lists the administrator's set of privileges. Create an instance of `Admin`, and call your method.

9-8. Privileges: Write a separate `Privileges` class. The class should have one attribute, `privileges`, that stores a list of strings as described in Exercise 9-7. Move the `show_privileges()` method to this class. Make a `Privileges` instance as an attribute in the `Admin` class. Create a new instance of `Admin` and use your method to show its privileges.

(continued)

9-9. Battery Upgrade: Use the final version of *electric_car.py* from this section.

Add a method to the *Battery* class called *upgrade_battery()*. This method should check the battery size and set the capacity to 100 if it isn't already.

Make an electric car with a default battery size, call *get_range()* once, and then call *get_range()* a second time after upgrading the battery. You should see an increase in the car's range.

Importing Classes

As you add more functionality to your classes, your files can get long, even when you use inheritance properly. In keeping with the overall philosophy of Python, you'll want to keep your files as uncluttered as possible. To help, Python lets you store classes in modules and then import the classes you need into your main program.

Importing a Single Class

Let's create a module containing just the *Car* class. This brings up a subtle naming issue: we already have a file named *car.py* in this chapter, but this module should be named *car.py* because it contains code representing a car. We'll resolve this naming issue by storing the *Car* class in a module named *car.py*, replacing the *car.py* file we were previously using. From now on, any program that uses this module will need a more specific filename, such as *my_car.py*. Here's *car.py* with just the code from the *Car*:

```
car.py ❶ """A class that can be used to represent a car."""

class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f'{self.year} {self.manufacturer} {self.model}'
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f'This car has {self.odometer_reading} miles on it.') 
```

```
def update_odometer(self, mileage):
    """
        Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back.
    """
    if mileage >= self.odometer_reading:
        self.odometer_reading = mileage
    else:
        print("You can't roll back an odometer!")

def increment_odometer(self, miles):
    """
        Add the given amount to the odometer reading.
    """
    self.odometer_reading += miles
```

At ❶ we include a module-level docstring that briefly describes the contents of this module. You should write a docstring for each module you create.

Now we make a separate file called *my_car.py*. This file will import the *Car* class and then create an instance from that class:

```
my_car.py ❶ from car import Car
```

```
my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

The import statement at ❶ tells Python to open the *car* module and import the class *Car*. Now we can use the *Car* class as if it were defined in this file. The output is the same as we saw earlier:

```
2019 Audi A4
This car has 23 miles on it.
```

Importing classes is an effective way to program. Picture how long this program file would be if the entire *Car* class were included. When you instead move the class to a module and import the module, you still get all the same functionality, but you keep your main program file clean and easy to read. You also store most of the logic in separate files; once your classes work as you want them to, you can leave those files alone and focus on the higher-level logic of your main program.

Storing Multiple Classes in a Module

You can store as many classes as you need in a single module, although each class in a module should be related somehow. The classes *Battery* and *ElectricCar* both help represent cars, so let's add them to the module *car.py*.

```
car.py     """A set of classes used to represent gas and electric cars."""

class Car:
    --snip--

class Battery:
    """A simple attempt to model a battery for an electric car."""

    def __init__(self, battery_size=70):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315

        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    """Models aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
        self.battery = Battery()
```

Now we can make a new file called *my_electric_car.py*, import the *ElectricCar* class, and make an electric car:

```
my_electric
    _car.py
from car import ElectricCar

my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

This has the same output we saw earlier, even though most of the logic is hidden away in a module:

```
2019 Tesla Model S
This car has a 75-kWh battery.
This car can go about 260 miles on a full charge.
```

Importing Multiple Classes from a Module

You can import as many classes as you need into a program file. If we want to make a regular car and an electric car in the same file, we need to import both classes, Car and ElectricCar:

```
my_cars.py ❶ from car import Car, ElectricCar  
  
❷ my_beetle = Car('volkswagen', 'beetle', 2019)  
print(my_beetle.get_descriptive_name())  
  
❸ my_tesla = ElectricCar('tesla', 'roadster', 2019)  
print(my_tesla.get_descriptive_name())
```

You import multiple classes from a module by separating each class with a comma ❶. Once you've imported the necessary classes, you're free to make as many instances of each class as you need.

In this example we make a regular Volkswagen Beetle at ❷ and an electric Tesla Roadster at ❸:

```
2019 Volkswagen Beetle  
2019 Tesla Roadster
```

Importing an Entire Module

You can also import an entire module and then access the classes you need using dot notation. This approach is simple and results in code that is easy to read. Because every call that creates an instance of a class includes the module name, you won't have naming conflicts with any names used in the current file.

Here's what it looks like to import the entire car module and then create a regular car and an electric car:

```
my_cars.py ❶ import car  
  
❷ my_beetle = car.Car('volkswagen', 'beetle', 2019)  
print(my_beetle.get_descriptive_name())  
  
❸ my_tesla = car.ElectricCar('tesla', 'roadster', 2019)  
print(my_tesla.get_descriptive_name())
```

At ❶ we import the entire car module. We then access the classes we need through the `module_name.ClassName` syntax. At ❷ we again create a Volkswagen Beetle, and at ❸ we create a Tesla Roadster.

Importing All Classes from a Module

You can import every class from a module using the following syntax:

```
from module_name import *
```

This method is not recommended for two reasons. First, it's helpful to be able to read the `import` statements at the top of a file and get a clear sense of which classes a program uses. With this approach it's unclear which classes you're using from the module. This approach can also lead to confusion with names in the file. If you accidentally import a class with the same name as something else in your program file, you can create errors that are hard to diagnose. I show this here because even though it's not a recommended approach, you're likely to see it in other people's code at some point.

If you need to import many classes from a module, you're better off importing the entire module and using the `module_name.ClassName` syntax. You won't see all the classes used at the top of the file, but you'll see clearly where the module is used in the program. You'll also avoid the potential naming conflicts that can arise when you import every class in a module.

Importing a Module into a Module

Sometimes you'll want to spread out your classes over several modules to keep any one file from growing too large and avoid storing unrelated classes in the same module. When you store your classes in several modules, you may find that a class in one module depends on a class in another module. When this happens, you can import the required class into the first module.

For example, let's store the `Car` class in one module and the `ElectricCar` and `Battery` classes in a separate module. We'll make a new module called `electric_car.py`—replacing the `electric_car.py` file we created earlier—and copy just the `Battery` and `ElectricCar` classes into this file:

```
electric_car.py     """A set of classes that can be used to represent electric cars."""
```

```
❶ from car import Car

class Battery:
    --snip--

class ElectricCar(Car):
    --snip--
```

The class `ElectricCar` needs access to its parent class `Car`, so we import `Car` directly into the module at ❶. If we forget this line, Python will raise an error when we try to import the `electric_car` module. We also need to update the `Car` module so it contains only the `Car` class:

```
car.py      """A class that can be used to represent a car."""

class Car:
    --snip--
```

Now we can import from each module separately and create whatever kind of car we need:

```
my_cars.py ❶ from car import Car
           from electric_car import ElectricCar

           my_beetle = Car('volkswagen', 'beetle', 2019)
           print(my_beetle.get_descriptive_name())

           my_tesla = ElectricCar('tesla', 'roadster', 2019)
           print(my_tesla.get_descriptive_name())
```

At ❶ we import `Car` from its module, and `ElectricCar` from its module. We then create one regular car and one electric car. Both kinds of cars are created correctly:

```
2019 Volkswagen Beetle
2019 Tesla Roadster
```

Using Aliases

As you saw in Chapter 8, aliases can be quite helpful when using modules to organize your projects' code. You can use aliases when importing classes as well.

As an example, consider a program where you want to make a bunch of electric cars. It might get tedious to type (and read) `ElectricCar` over and over again. You can give `ElectricCar` an alias in the import statement:

```
from electric_car import ElectricCar as EC
```

Now you can use this alias whenever you want to make an electric car:

```
my_tesla = EC('tesla', 'roadster', 2019)
```

Finding Your Own Workflow

As you can see, Python gives you many options for how to structure code in a large project. It's important to know all these possibilities so you can determine the best ways to organize your projects as well as understand other people's projects.

When you're starting out, keep your code structure simple. Try doing everything in one file and moving your classes to separate modules once everything is working. If you like how modules and files interact, try storing your classes in modules when you start a project. Find an approach that lets you write code that works, and go from there.

TRY IT YOURSELF

9-10. Imported Restaurant: Using your latest Restaurant class, store it in a module. Make a separate file that imports Restaurant. Make a Restaurant instance, and call one of Restaurant's methods to show that the `import` statement is working properly.

9-11. Imported Admin: Start with your work from Exercise 9-8 (page 173). Store the classes User, Privileges, and Admin in one module. Create a separate file, make an Admin instance, and call `show_privileges()` to show that everything is working correctly.

9-12. Multiple Modules: Store the User class in one module, and store the Privileges and Admin classes in a separate module. In a separate file, create an Admin instance and call `show_privileges()` to show that everything is still working correctly.

The Python Standard Library

The *Python standard library* is a set of modules included with every Python installation. Now that you have a basic understanding of how functions and classes work, you can start to use modules like these that other programmers have written. You can use any function or class in the standard library by including a simple `import` statement at the top of your file. Let's look at one module, `random`, which can be useful in modeling many real-world situations.

One interesting function from the `random` module is `randint()`. This function takes two integer arguments and returns a randomly selected integer between (and including) those numbers.

Here's how to generate a random number between 1 and 6:

```
>>> from random import randint
>>> randint(1, 6)
3
```

Another useful function is `choice()`. This function takes in a list or tuple and returns a randomly chosen element:

```
>>> from random import choice
>>> players = ['charles', 'martina', 'michael', 'florence', 'eli']
>>> first_up = choice(players)
>>> first_up
'florence'
```

The `random` module shouldn't be used when building security-related applications, but it's good enough for many fun and interesting projects.

NOTE

You can also download modules from external sources. You'll see a number of these examples in Part II, where we'll need external modules to complete each project.

TRY IT YOURSELF

9-13. Dice: Make a class `Die` with one attribute called `sides`, which has a default value of 6. Write a method called `roll_die()` that prints a random number between 1 and the number of sides the die has. Make a 6-sided die and roll it 10 times.

Make a 10-sided die and a 20-sided die. Roll each die 10 times.

9-14. Lottery: Make a list or tuple containing a series of 10 numbers and five letters. Randomly select four numbers or letters from the list and print a message saying that any ticket matching these four numbers or letters wins a prize.

9-15. Lottery Analysis: You can use a loop to see how hard it might be to win the kind of lottery you just modeled. Make a list or tuple called `my_ticket`. Write a loop that keeps pulling numbers until your ticket wins. Print a message reporting how many times the loop had to run to give you a winning ticket.

9-16. Python Module of the Week: One excellent resource for exploring the Python standard library is a site called *Python Module of the Week*. Go to <https://pymotw.com/> and look at the table of contents. Find a module that looks interesting to you and read about it, perhaps starting with the `random` module.

Styling Classes

A few styling issues related to classes are worth clarifying, especially as your programs become more complicated.

Class names should be written in *CamelCase*. To do this, capitalize the first letter of each word in the name, and don't use underscores. Instance and module names should be written in lowercase with underscores between words.

Every class should have a docstring immediately following the class definition. The docstring should be a brief description of what the class does, and you should follow the same formatting conventions you used for writing docstrings in functions. Each module should also have a docstring describing what the classes in a module can be used for.

You can use blank lines to organize code, but don't use them excessively. Within a class you can use one blank line between methods, and within a module you can use two blank lines to separate classes.

If you need to import a module from the standard library and a module that you wrote, place the import statement for the standard library module first. Then add a blank line and the import statement for the module you wrote. In programs with multiple import statements, this convention makes it easier to see where the different modules used in the program come from.

Summary

In this chapter you learned how to write your own classes. You learned how to store information in a class using attributes and how to write methods that give your classes the behavior they need. You learned to write `__init__()` methods that create instances from your classes with exactly the attributes you want. You saw how to modify the attributes of an instance directly and through methods. You learned that inheritance can simplify the creation of classes that are related to each other, and you learned to use instances of one class as attributes in another class to keep each class simple.

You saw how storing classes in modules and importing classes you need into the files where they'll be used can keep your projects organized. You started learning about the Python standard library, and you saw an example based on the `random` module. Finally, you learned to style your classes using Python conventions.

In Chapter 10 you'll learn to work with files so you can save the work you've done in a program and the work you've allowed users to do. You'll also learn about *exceptions*, a special Python class designed to help you respond to errors when they arise.

10

FILES AND EXCEPTIONS



Now that you've mastered the basic skills you need to write organized programs that are easy to use, it's time to think about making your programs even more relevant and usable. In this chapter you'll learn to work with files so your programs can quickly analyze lots of data.

You'll learn to handle errors so your programs don't crash when they encounter unexpected situations. You'll learn about *exceptions*, which are special objects Python creates to manage errors that arise while a program is running. You'll also learn about the `json` module, which allows you to save user data so it isn't lost when your program stops running.

Learning to work with files and save data will make your programs easier for people to use. Users will be able to choose what data to enter and when to enter it. People can run your program, do some work, and then close the program and pick up where they left off later. Learning to handle exceptions will help you deal with situations in which files don't exist and deal with other problems that can cause your programs to crash. This will make your programs more robust when they encounter bad data, whether

it comes from innocent mistakes or from malicious attempts to break your programs. With the skills you'll learn in this chapter, you'll make your programs more applicable, usable, and stable.

Reading from a File

An incredible amount of data is available in text files. Text files can contain weather data, traffic data, socioeconomic data, literary works, and more. Reading from a file is particularly useful in data analysis applications, but it's also applicable to any situation in which you want to analyze or modify information stored in a file. For example, you can write a program that reads in the contents of a text file and rewrites the file with formatting that allows a browser to display it.

When you want to work with the information in a text file, the first step is to read the file into memory. You can read the entire contents of a file, or you can work through the file one line at a time.

Reading an Entire File

To begin, we need a file with a few lines of text in it. Let's start with a file that contains *pi* to 30 decimal places, with 10 decimal places per line:

<i>pi_digits.txt</i>	3.1415926535 8979323846 2643383279
----------------------	------------------------------------------

To try the following examples yourself, you can enter these lines in an editor and save the file as *pi_digits.txt*, or you can download the file from the book's resources through <https://nostarch.com/pythoncrashcourse2e/>. Save the file in the same directory where you'll store this chapter's programs.

Here's a program that opens this file, reads it, and prints the contents of the file to the screen:

<i>file_reader.py</i>	with open('pi_digits.txt') as file_object: contents = file_object.read() print(contents)
-----------------------	------------------------------------------------------------------------------------------------

The first line of this program has a lot going on. Let's start by looking at the `open()` function. To do any work with a file, even just printing its contents, you first need to *open* the file to access it. The `open()` function needs one argument: the name of the file you want to open. Python looks for this file in the directory where the program that's currently being executed is stored. In this example, *file_reader.py* is currently running, so Python looks for *pi_digits.txt* in the directory where *file_reader.py* is stored. The `open()` function returns an object representing the file. Here, `open('pi_digits.txt')` returns an object representing *pi_digits.txt*. Python assigns this object to `file_object`, which we'll work with later in the program.

The keyword `with` closes the file once access to it is no longer needed. Notice how we call `open()` in this program but not `close()`. You could open and close the file by calling `open()` and `close()`, but if a bug in your program prevents the `close()` method from being executed, the file may never close. This may seem trivial, but improperly closed files can cause data to be lost or corrupted. And if you call `close()` too early in your program, you'll find yourself trying to work with a *closed* file (a file you can't access), which leads to more errors. It's not always easy to know exactly when you should close a file, but with the structure shown here, Python will figure that out for you. All you have to do is open the file and work with it as desired, trusting that Python will close it automatically when the `with` block finishes execution.

Once we have a file object representing `pi_digits.txt`, we use the `read()` method in the second line of our program to read the entire contents of the file and store it as one long string in `contents`. When we print the value of `contents`, we get the entire text file back:

```
3.1415926535  
8979323846  
2643383279
```

The only difference between this output and the original file is the extra blank line at the end of the output. The blank line appears because `read()` returns an empty string when it reaches the end of the file; this empty string shows up as a blank line. If you want to remove the extra blank line, you can use `rstrip()` in the call to `print()`:

```
with open('pi_digits.txt') as file_object:  
    contents = file_object.read()  
    print(contents.rstrip())
```

Recall that Python's `rstrip()` method removes, or strips, any whitespace characters from the right side of a string. Now the output matches the contents of the original file exactly:

```
3.1415926535  
8979323846  
2643383279
```

File Paths

When you pass a simple filename like `pi_digits.txt` to the `open()` function, Python looks in the directory where the file that's currently being executed (that is, your `.py` program file) is stored.

Sometimes, depending on how you organize your work, the file you want to open won't be in the same directory as your program file. For example, you might store your program files in a folder called

python_work; inside *python_work*, you might have another folder called *text_files* to distinguish your program files from the text files they’re manipulating. Even though *text_files* is in *python_work*, just passing `open()` the name of a file in *text_files* won’t work, because Python will only look in *python_work* and stop there; it won’t go on and look in *text_files*. To get Python to open files from a directory other than the one where your program file is stored, you need to provide a *file path*, which tells Python to look in a specific location on your system.

Because *text_files* is inside *python_work*, you could use a relative file path to open a file from *text_files*. A *relative file path* tells Python to look for a given location relative to the directory where the currently running program file is stored. For example, you’d write:

```
with open('text_files/filename.txt') as file_object:
```

This line tells Python to look for the desired `.txt` file in the folder *text_files* and assumes that *text_files* is located inside *python_work* (which it is).

NOTE

Windows systems use a backslash (\) instead of a forward slash (/) when displaying file paths, but you can still use forward slashes in your code.

You can also tell Python exactly where the file is on your computer regardless of where the program that’s being executed is stored. This is called an *absolute file path*. You use an absolute path if a relative path doesn’t work. For instance, if you’ve put *text_files* in some folder other than *python_work*—say, a folder called *other_files*—then just passing `open()` the path `'text_files/filename.txt'` won’t work because Python will only look for that location inside *python_work*. You’ll need to write out a full path to clarify where you want Python to look.

Absolute paths are usually longer than relative paths, so it’s helpful to assign them to a variable and then pass that variable to `open()`:

```
file_path = '/home/ehmatthes/other_files/text_files/filename.txt'  
with open(file_path) as file_object:
```

Using absolute paths, you can read files from any location on your system. For now it’s easiest to store files in the same directory as your program files or in a folder such as *text_files* within the directory that stores your program files.

NOTE

If you try to use backslashes in a file path, you’ll get an error because the backslash is used to escape characters in strings. For example, in the path "C:\path\to\file.txt", the sequence \t is interpreted as a tab. If you need to use backslashes, you can escape each one in the path, like this: "C:\\path\\\\to\\\\file.txt".

Reading Line by Line

When you’re reading a file, you’ll often want to examine each line of the file. You might be looking for certain information in the file, or you might want to modify the text in the file in some way. For example, you might want to read through a file of weather data and work with any line that includes the word *sunny* in the description of that day’s weather. In a news report, you might look for any line with the tag `<headline>` and rewrite that line with a specific kind of formatting.

You can use a `for` loop on the file object to examine each line from a file one at a time:

```
file_reader.py ❶ filename = 'pi_digits.txt'

❷ with open(filename) as file_object:
❸     for line in file_object:
        print(line)
```

At ❶ we assign the name of the file we’re reading from to the variable `filename`. This is a common convention when working with files. Because the variable `filename` doesn’t represent the actual file—it’s just a string telling Python where to find the file—you can easily swap out `'pi_digits.txt'` for the name of another file you want to work with. After we call `open()`, an object representing the file and its contents is assigned to the variable `file_object` ❷. We again use the `with` syntax to let Python open and close the file properly. To examine the file’s contents, we work through each line in the file by looping over the file object ❸.

When we print each line, we find even more blank lines:

```
3.1415926535

8979323846

2643383279
```

These blank lines appear because an invisible newline character is at the end of each line in the text file. The `print` function adds its own newline each time we call it, so we end up with two newline characters at the end of each line: one from the file and one from `print()`. Using `rstrip()` on each line in the `print()` call eliminates these extra blank lines:

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
    for line in file_object:
        print(line.rstrip())
```

Now the output matches the contents of the file once again:

```
3.1415926535  
8979323846  
2643383279
```

Making a List of Lines from a File

When you use `with`, the file object returned by `open()` is only available inside the `with` block that contains it. If you want to retain access to a file's contents outside the `with` block, you can store the file's lines in a list inside the `with` block and then work with that list. You can process parts of the file immediately and postpone some processing for later in the program.

The following example stores the lines of `pi_digits.txt` in a list inside the `with` block and then prints the lines outside the `with` block:

```
filename = 'pi_digits.txt'  
  
with open(filename) as file_object:  
❶    lines = file_object.readlines()  
  
❷ for line in lines:  
    print(line.rstrip())
```

At ❶ the `readlines()` method takes each line from the file and stores it in a list. This list is then assigned to `lines`, which we can continue to work with after the `with` block ends. At ❷ we use a simple `for` loop to print each line from `lines`. Because each item in `lines` corresponds to each line in the file, the output matches the contents of the file exactly.

Working with a File's Contents

After you've read a file into memory, you can do whatever you want with that data, so let's briefly explore the digits of `pi`. First, we'll attempt to build a single string containing all the digits in the file with no whitespace in it:

```
pi_string.py  
  
filename = 'pi_digits.txt'  
  
with open(filename) as file_object:  
    lines = file_object.readlines()  
  
❶ pi_string = ''  
❷ for line in lines:  
    pi_string += line.rstrip()  
  
❸ print(pi_string)  
print(len(pi_string))
```

We start by opening the file and storing each line of digits in a list, just as we did in the previous example. At ❶ we create a variable, `pi_string`, to hold the digits of `pi`. We then create a loop that adds each line of digits to `pi_string` and removes the newline character from each line ❷. At ❸ we print this string and also show how long the string is:

```
3.1415926535 8979323846 2643383279
```

```
36
```

The variable `pi_string` contains the whitespace that was on the left side of the digits in each line, but we can get rid of that by using `strip()` instead of `rstrip()`:

```
--snip--  
for line in lines:  
    pi_string += line.strip()  
  
print(pi_string)  
print(len(pi_string))
```

Now we have a string containing `pi` to 30 decimal places. The string is 32 characters long because it also includes the leading 3 and a decimal point:

```
3.141592653589793238462643383279
```

```
32
```

NOTE

When Python reads from a text file, it interprets all text in the file as a string. If you read in a number and want to work with that value in a numerical context, you'll have to convert it to an integer using the `int()` function or convert it to a float using the `float()` function.

Large Files: One Million Digits

So far we've focused on analyzing a text file that contains only three lines, but the code in these examples would work just as well on much larger files. If we start with a text file that contains `pi` to 1,000,000 decimal places instead of just 30, we can create a single string containing all these digits. We don't need to change our program at all except to pass it a different file. We'll also print just the first 50 decimal places, so we don't have to watch a million digits scroll by in the terminal:

```
pi_string.py  
filename = 'pi_million_digits.txt'  
  
with open(filename) as file_object:  
    lines = file_object.readlines()
```

```
pi_string = ''  
for line in lines:  
    pi_string += line.strip()  
  
print(f"{pi_string[:52]}...")  
print(len(pi_string))
```

The output shows that we do indeed have a string containing *pi* to 1,000,000 decimal places:

```
3.14159265358979323846264338327950288419716939937510...  
1000002
```

Python has no inherent limit to how much data you can work with; you can work with as much data as your system's memory can handle.

NOTE

To run this program (and many of the examples that follow), you'll need to download the resources available at <https://nostarch.com/pythoncrashcourse2e/>.

Is Your Birthday Contained in Pi?

I've always been curious to know if my birthday appears anywhere in the digits of *pi*. Let's use the program we just wrote to find out if someone's birthday appears anywhere in the first million digits of *pi*. We can do this by expressing each birthday as a string of digits and seeing if that string appears anywhere in *pi_string*:

```
--snip--  
for line in lines:  
    pi_string += line.strip()  
  
❶ birthday = input("Enter your birthday, in the form mmddyy: ")  
❷ if birthday in pi_string:  
    print("Your birthday appears in the first million digits of pi!")  
else:  
    print("Your birthday does not appear in the first million digits of pi.")
```

At ❶ we prompt for the user's birthday, and then at ❷ we check if that string is in *pi_string*. Let's try it:

```
Enter your birthdate, in the form mmddyy: 120372  
Your birthday appears in the first million digits of pi!
```

My birthday does appear in the digits of *pi*! Once you've read from a file, you can analyze its contents in just about any way you can imagine.

TRY IT YOURSELF

10-1. Learning Python: Open a blank file in your text editor and write a few lines summarizing what you've learned about Python so far. Start each line with the phrase *In Python you can. . .*. Save the file as *learning_python.txt* in the same directory as your exercises from this chapter. Write a program that reads the file and prints what you wrote three times. Print the contents once by reading in the entire file, once by looping over the file object, and once by storing the lines in a list and then working with them outside the *with* block.

10-2. Learning C: You can use the *replace()* method to replace any word in a string with a different word. Here's a quick example showing how to replace 'dog' with 'cat' in a sentence:

```
>>> message = "I really like dogs."  
>>> message.replace('dog', 'cat')  
'I really like cats.'
```

Read in each line from the file you just created, *learning_python.txt*, and replace the word *Python* with the name of another language, such as *C*. Print each modified line to the screen.

Writing to a File

One of the simplest ways to save data is to write it to a file. When you write text to a file, the output will still be available after you close the terminal containing your program's output. You can examine output after a program finishes running, and you can share the output files with others as well. You can also write programs that read the text back into memory and work with it again later.

Writing to an Empty File

To write text to a file, you need to call *open()* with a second argument telling Python that you want to write to the file. To see how this works, let's write a simple message and store it in a file instead of printing it to the screen:

```
write  
_message.py  
filename = 'programming.txt'  
❶ with open(filename, 'w') as file_object:  
❷     file_object.write("I love programming.")
```

The call to *open()* in this example has two arguments ❶. The first argument is still the name of the file we want to open. The second argument, '*w*', tells Python that we want to open the file in *write mode*. You can open a file

in *read mode* ('r'), *write mode* ('w'), *append mode* ('a'), or a mode that allows you to read and write to the file ('r+'). If you omit the mode argument, Python opens the file in read-only mode by default.

The `open()` function automatically creates the file you're writing to if it doesn't already exist. However, be careful opening a file in write mode ('w') because if the file does exist, Python will erase the contents of the file before returning the file object.

At ❷ we use the `write()` method on the file object to write a string to the file. This program has no terminal output, but if you open the file `programming.txt`, you'll see one line:

`programming.txt` I love programming.

This file behaves like any other file on your computer. You can open it, write new text in it, copy from it, paste to it, and so forth.

NOTE

Python can only write strings to a text file. If you want to store numerical data in a text file, you'll have to convert the data to string format first using the `str()` function.

Writing Multiple Lines

The `write()` function doesn't add any newlines to the text you write. So if you write more than one line without including newline characters, your file may not look the way you want it to:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
    file_object.write("I love creating new games.")
```

If you open `programming.txt`, you'll see the two lines squished together:

I love programming.I love creating new games.

Including newlines in your calls to `write()` makes each string appear on its own line:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.\n")
    file_object.write("I love creating new games.\n")
```

The output now appears on separate lines:

I love programming.
I love creating new games.

You can also use spaces, tab characters, and blank lines to format your output, just as you've been doing with terminal-based output.

Appending to a File

If you want to add content to a file instead of writing over existing content, you can open the file in *append mode*. When you open a file in append mode, Python doesn't erase the contents of the file before returning the file object. Any lines you write to the file will be added at the end of the file. If the file doesn't exist yet, Python will create an empty file for you.

Let's modify *write_message.py* by adding some new reasons we love programming to the existing file *programming.txt*:

```
write
message.py
filename = 'programming.txt'
❶ with open(filename, 'a') as file_object:
❷     file_object.write("I also love finding meaning in large datasets.\n")
        file_object.write("I love creating apps that can run in a browser.\n")
```

At ❶ we use the '`a`' argument to open the file for appending rather than writing over the existing file. At ❷ we write two new lines, which are added to *programming.txt*:

```
programming.txt
I love programming.
I love creating new games.
I also love finding meaning in large datasets.
I love creating apps that can run in a browser.
```

We end up with the original contents of the file, followed by the new content we just added.

TRY IT YOURSELF

10-3. Guest: Write a program that prompts the user for their name. When they respond, write their name to a file called *guest.txt*.

10-4. Guest Book: Write a while loop that prompts users for their name. When they enter their name, print a greeting to the screen and add a line recording their visit in a file called *guest_book.txt*. Make sure each entry appears on a new line in the file.

10-5. Programming Poll: Write a while loop that asks people why they like programming. Each time someone enters a reason, add their reason to a file that stores all the responses.

Exceptions

Python uses special objects called *exceptions* to manage errors that arise during a program's execution. Whenever an error occurs that makes Python unsure what to do next, it creates an exception object. If you write code that handles the exception, the program will continue running. If you don't handle the exception, the program will halt and show a *traceback*, which includes a report of the exception that was raised.

Exceptions are handled with try-except blocks. A try-except block asks Python to do something, but it also tells Python what to do if an exception is raised. When you use try-except blocks, your programs will continue running even if things start to go wrong. Instead of tracebacks, which can be confusing for users to read, users will see friendly error messages that you write.

Handling the ZeroDivisionError Exception

Let's look at a simple error that causes Python to raise an exception. You probably know that it's impossible to divide a number by zero, but let's ask Python to do it anyway:

```
division  
calculator.py
```

```
print(5/0)
```

Of course Python can't do this, so we get a traceback:

```
Traceback (most recent call last):  
  File "division_calculator.py", line 1, in <module>  
    print(5/0)  
❶ ZeroDivisionError: division by zero
```

The error reported at ❶ in the traceback, `ZeroDivisionError`, is an exception object. Python creates this kind of object in response to a situation where it can't do what we ask it to. When this happens, Python stops the program and tells us the kind of exception that was raised. We can use this information to modify our program. We'll tell Python what to do when this kind of exception occurs; that way, if it happens again, we're prepared.

Using try-except Blocks

When you think an error may occur, you can write a try-except block to handle the exception that might be raised. You tell Python to try running some code, and you tell it what to do if the code results in a particular kind of exception.

Here's what a try-except block for handling the `ZeroDivisionError` exception looks like:

```
try:  
    print(5/0)  
except ZeroDivisionError:  
    print("You can't divide by zero!")
```

We put `print(5/0)`, the line that caused the error, inside a `try` block. If the code in a `try` block works, Python skips over the `except` block. If the code in the `try` block causes an error, Python looks for an `except` block whose error matches the one that was raised and runs the code in that block.

In this example, the code in the `try` block produces a `ZeroDivisionError`, so Python looks for an `except` block telling it how to respond. Python then runs the code in that block, and the user sees a friendly error message instead of a traceback:

```
You can't divide by zero!
```

If more code followed the `try`-`except` block, the program would continue running because we told Python how to handle the error. Let's look at an example where catching an error can allow a program to continue running.

Using Exceptions to Prevent Crashes

Handling errors correctly is especially important when the program has more work to do after the error occurs. This happens often in programs that prompt users for input. If the program responds to invalid input appropriately, it can prompt for more valid input instead of crashing.

Let's create a simple calculator that does only division:

```
division
calculator.py

print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
    ❶    first_number = input("\nFirst number: ")
        if first_number == 'q':
            break
    ❷    second_number = input("Second number: ")
        if second_number == 'q':
            break
    ❸    answer = int(first_number) / int(second_number)
    print(answer)
```

This program prompts the user to input a `first_number` ❶ and, if the user does not enter `q` to quit, a `second_number` ❷. We then divide these two numbers to get an `answer` ❸. This program does nothing to handle errors, so asking it to divide by zero causes it to crash:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.
```

```
First number: 5
Second number: 0
Traceback (most recent call last):
  File "division_calculator.py", line 9, in <module>
    answer = int(first_number) / int(second_number)
ZeroDivisionError: division by zero
```

It's bad that the program crashed, but it's also not a good idea to let users see tracebacks. Nontechnical users will be confused by them, and in a malicious setting, attackers will learn more than you want them to know from a traceback. For example, they'll know the name of your program file, and they'll see a part of your code that isn't working properly. A skilled attacker can sometimes use this information to determine which kind of attacks to use against your code.

The `else` Block

We can make this program more error resistant by wrapping the line that might produce errors in a try-except block. The error occurs on the line that performs the division, so that's where we'll put the try-except block. This example also includes an `else` block. Any code that depends on the try block executing successfully goes in the `else` block:

```
--snip--
while True:
    --snip--
    if second_number == 'q':
        break
❶    try:
❷        answer = int(first_number) / int(second_number)
❸    except ZeroDivisionError:
❹        print("You can't divide by 0!")
❺    else:
❻        print(answer)
```

We ask Python to try to complete the division operation in a try block ❶, which includes only the code that might cause an error. Any code that depends on the try block succeeding is added to the `else` block. In this case if the division operation is successful, we use the `else` block to print the result ❺.

The `except` block tells Python how to respond when a `ZeroDivisionError` arises ❷. If the try block doesn't succeed because of a division by zero error, we print a friendly message telling the user how to avoid this kind of error. The program continues to run, and the user never sees a traceback:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.
```

```
First number: 5
Second number: 0
You can't divide by 0!
```

```
First number: 5
Second number: 2
2.5
```

```
First number: q
```

The try-except-else block works like this: Python attempts to run the code in the try block. The only code that should go in a try block is code that might cause an exception to be raised. Sometimes you'll have additional code that should run only if the try block was successful; this code goes in the else block. The except block tells Python what to do in case a certain exception arises when it tries to run the code in the try block.

By anticipating likely sources of errors, you can write robust programs that continue to run even when they encounter invalid data and missing resources. Your code will be resistant to innocent user mistakes and malicious attacks.

Handling the FileNotFoundError Exception

One common issue when working with files is handling missing files. The file you're looking for might be in a different location, the filename may be misspelled, or the file may not exist at all. You can handle all of these situations in a straightforward way with a try-except block.

Let's try to read a file that doesn't exist. The following program tries to read in the contents of *Alice in Wonderland*, but I haven't saved the file *alice.txt* in the same directory as *alice.py*:

```
alice.py
filename = 'alice.txt'

with open(filename, encoding='utf-8') as f:
    contents = f.read()
```

There are two changes here. One is the use of the variable *f* to represent the file object, which is a common convention. The second is the use of the *encoding* argument. This argument is needed when your system's default encoding doesn't match the encoding of the file that's being read.

Python can't read from a missing file, so it raises an exception:

```
Traceback (most recent call last):
  File "alice.py", line 3, in <module>
    with open(filename, encoding='utf-8') as f:
FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'
```

The last line of the traceback reports a *FileNotFoundException*: this is the exception Python creates when it can't find the file it's trying to open.

In this example, the `open()` function produces the error, so to handle it, the `try` block will begin with the line that contains `open()`:

```
filename = 'alice.txt'

try:
    with open(filename, encoding='utf-8') as f:
        contents = f.read()
except FileNotFoundError:
    print(f"Sorry, the file {filename} does not exist.")
```

In this example, the code in the `try` block produces a `FileNotFoundException`, so Python looks for an `except` block that matches that error. Python then runs the code in that block, and the result is a friendly error message instead of a traceback:

```
Sorry, the file alice.txt does not exist.
```

The program has nothing more to do if the file doesn't exist, so the error-handling code doesn't add much to this program. Let's build on this example and see how exception handling can help when you're working with more than one file.

Analyzing Text

You can analyze text files containing entire books. Many classic works of literature are available as simple text files because they are in the public domain. The texts used in this section come from Project Gutenberg (<http://gutenberg.org/>). Project Gutenberg maintains a collection of literary works that are available in the public domain, and it's a great resource if you're interested in working with literary texts in your programming projects.

Let's pull in the text of *Alice in Wonderland* and try to count the number of words in the text. We'll use the string method `split()`, which can build a list of words from a string. Here's what `split()` does with a string containing just the title "Alice in Wonderland":

```
>>> title = "Alice in Wonderland"
>>> title.split()
['Alice', 'in', 'Wonderland']
```

The `split()` method separates a string into parts wherever it finds a space and stores all the parts of the string in a list. The result is a list of words from the string, although some punctuation may also appear with some of the words. To count the number of words in *Alice in Wonderland*, we'll use `split()` on the entire text. Then we'll count the items in the list to get a rough idea of the number of words in the text:

```
filename = 'alice.txt'

try:
```

```
        with open(filename, encoding='utf-8') as f:
            contents = f.read()
    except FileNotFoundError:
        print(f"Sorry, the file {filename} does not exist.")
    else:
        # Count the approximate number of words in the file.
❶    words = contents.split()
❷    num_words = len(words)
❸    print(f"The file {filename} has about {num_words} words.")
```

I moved the file *alice.txt* to the correct directory, so the try block will work this time. At ❶ we take the string `contents`, which now contains the entire text of *Alice in Wonderland* as one long string, and use the `split()` method to produce a list of all the words in the book. When we use `len()` on this list to examine its length, we get a good approximation of the number of words in the original string ❷. At ❸ we print a statement that reports how many words were found in the file. This code is placed in the `else` block because it will work only if the code in the try block was executed successfully. The output tells us how many words are in *alice.txt*:

The file alice.txt has about 29465 words.

The count is a little high because extra information is provided by the publisher in the text file used here, but it's a good approximation of the length of *Alice in Wonderland*.

Working with Multiple Files

Let's add more books to analyze. But before we do, let's move the bulk of this program to a function called `count_words()`. By doing so, it will be easier to run the analysis for multiple books:

```
word_count.py  def count_words(filename):
❶    """Count the approximate number of words in a file."""
    try:
        with open(filename, encoding='utf-8') as f:
            contents = f.read()
    except FileNotFoundError:
        print(f"Sorry, the file {filename} does not exist.")
    else:
        words = contents.split()
        num_words = len(words)
        print(f"The file {filename} has about {num_words} words.")

filename = 'alice.txt'
count_words(filename)
```

Most of this code is unchanged. We simply indented it and moved it into the body of `count_words()`. It's a good habit to keep comments up to date when you're modifying a program, so we changed the comment to a doc-string and reworded it slightly ❶.

Now we can write a simple loop to count the words in any text we want to analyze. We do this by storing the names of the files we want to analyze in a list, and then we call `count_words()` for each file in the list. We'll try to count the words for *Alice in Wonderland*, *Siddhartha*, *Moby Dick*, and *Little Women*, which are all available in the public domain. I've intentionally left *siddhartha.txt* out of the directory containing *word_count.py*, so we can see how well our program handles a missing file:

```
def count_words(filename):
    --snip--

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
for filename in filenames:
    count_words(filename)
```

The missing *siddhartha.txt* file has no effect on the rest of the program's execution:

```
The file alice.txt has about 29465 words.
Sorry, the file siddhartha.txt does not exist.
The file moby_dick.txt has about 215830 words.
The file little_women.txt has about 189079 words.
```

Using the `try-except` block in this example provides two significant advantages. We prevent our users from seeing a traceback, and we let the program continue analyzing the texts it's able to find. If we don't catch the `FileNotFoundException` that *siddhartha.txt* raised, the user would see a full traceback, and the program would stop running after trying to analyze *Siddhartha*. It would never analyze *Moby Dick* or *Little Women*.

Failing Silently

In the previous example, we informed our users that one of the files was unavailable. But you don't need to report every exception you catch. Sometimes you'll want the program to fail silently when an exception occurs and continue on as if nothing happened. To make a program fail silently, you write a `try` block as usual, but you explicitly tell Python to do nothing in the `except` block. Python has a `pass` statement that tells it to do nothing in a block:

```
def count_words(filename):
    """Count the approximate number of words in a file."""
    try:
        --snip--
    except FileNotFoundError:
        ①     pass
    else:
        --snip--

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
for filename in filenames:
    count_words(filename)
```

The only difference between this listing and the previous one is the `pass` statement at ❶. Now when a `FileNotFoundException` is raised, the code in the `except` block runs, but nothing happens. No traceback is produced, and there's no output in response to the error that was raised. Users see the word counts for each file that exists, but they don't see any indication that a file wasn't found:

```
The file alice.txt has about 29465 words.  
The file moby_dick.txt has about 21580 words.  
The file little_women.txt has about 189079 words.
```

The `pass` statement also acts as a placeholder. It's a reminder that you're choosing to do nothing at a specific point in your program's execution and that you might want to do something there later. For example, in this program we might decide to write any missing filenames to a file called `missing_files.txt`. Our users wouldn't see this file, but we'd be able to read the file and deal with any missing texts.

Deciding Which Errors to Report

How do you know when to report an error to your users and when to fail silently? If users know which texts are supposed to be analyzed, they might appreciate a message informing them why some texts were not analyzed. If users expect to see some results but don't know which books are supposed to be analyzed, they might not need to know that some texts were unavailable. Giving users information they aren't looking for can decrease the usability of your program. Python's error-handling structures give you fine-grained control over how much to share with users when things go wrong; it's up to you to decide how much information to share.

Well-written, properly tested code is not very prone to internal errors, such as syntax or logical errors. But every time your program depends on something external, such as user input, the existence of a file, or the availability of a network connection, there is a possibility of an exception being raised. A little experience will help you know where to include exception handling blocks in your program and how much to report to users about errors that arise.

TRY IT YOURSELF

10-6. Addition: One common problem when prompting for numerical input occurs when people provide text instead of numbers. When you try to convert the input to an `int`, you'll get a `ValueError`. Write a program that prompts for two numbers. Add them together and print the result. Catch the `ValueError` if either input value is not a number, and print a friendly error message. Test your program by entering two numbers and then by entering some text instead of a number.

(continued)

10-7. Addition Calculator: Wrap your code from Exercise 10-6 in a while loop so the user can continue entering numbers even if they make a mistake and enter text instead of a number.

10-8. Cats and Dogs: Make two files, *cats.txt* and *dogs.txt*. Store at least three names of cats in the first file and three names of dogs in the second file. Write a program that tries to read these files and print the contents of the file to the screen. Wrap your code in a try-except block to catch the `FileNotFoundException`, and print a friendly message if a file is missing. Move one of the files to a different location on your system, and make sure the code in the except block executes properly.

10-9. Silent Cats and Dogs: Modify your except block in Exercise 10-8 to fail silently if either file is missing.

10-10. Common Words: Visit Project Gutenberg (<https://gutenberg.org/>) and find a few texts you'd like to analyze. Download the text files for these works, or copy the raw text from your browser into a text file on your computer.

You can use the `count()` method to find out how many times a word or phrase appears in a string. For example, the following code counts the number of times 'row' appears in a string:

```
>>> line = "Row, row, row your boat"
>>> line.count('row')
2
>>> line.lower().count('row')
3
```

Notice that converting the string to lowercase using `lower()` catches all appearances of the word you're looking for, regardless of how it's formatted.

Write a program that reads the files you found at Project Gutenberg and determines how many times the word 'the' appears in each text. This will be an approximation because it will also count words such as 'then' and 'there'. Try counting 'the ', with a space in the string, and see how much lower your count is.

Storing Data

Many of your programs will ask users to input certain kinds of information. You might allow users to store preferences in a game or provide data for a visualization. Whatever the focus of your program is, you'll store the information users provide in data structures such as lists and dictionaries. When users close a program, you'll almost always want to save the information they entered. A simple way to do this involves storing your data using the `json` module.

The `json` module allows you to dump simple Python data structures into a file and load the data from that file the next time the program runs. You can also use `json` to share data between different Python programs. Even better, the JSON data format is not specific to Python, so you can share data you store in the JSON format with people who work in many other programming languages. It's a useful and portable format, and it's easy to learn.

NOTE

The JSON (JavaScript Object Notation) format was originally developed for JavaScript. However, it has since become a common format used by many languages, including Python.

Using `json.dump()` and `json.load()`

Let's write a short program that stores a set of numbers and another program that reads these numbers back into memory. The first program will use `json.dump()` to store the set of numbers, and the second program will use `json.load()`.

The `json.dump()` function takes two arguments: a piece of data to store and a file object it can use to store the data. Here's how you can use `json.dump()` to store a list of numbers:

```
number
writer.py
import json
numbers = [2, 3, 5, 7, 11, 13]
❶ filename = 'numbers.json'
❷ with open(filename, 'w') as f:
❸     json.dump(numbers, f)
```

We first import the `json` module and then create a list of numbers to work with. At ❶ we choose a filename in which to store the list of numbers. It's customary to use the file extension `.json` to indicate that the data in the file is stored in the JSON format. Then we open the file in write mode, which allows `json` to write the data to the file ❷. At ❸ we use the `json.dump()` function to store the list `numbers` in the file `numbers.json`.

This program has no output, but let's open the file `numbers.json` and look at it. The data is stored in a format that looks just like Python:

```
[2, 3, 5, 7, 11, 13]
```

Now we'll write a program that uses `json.load()` to read the list back into memory:

```
number
reader.py
import json
❶ filename = 'numbers.json'
❷ with open(filename) as f:
❸     numbers = json.load(f)
print(numbers)
```

At ❶ we make sure to read from the same file we wrote to. This time when we open the file, we open it in read mode because Python only needs to read from the file ❷. At ❸ we use the `json.load()` function to load the information stored in `numbers.json`, and we assign it to the variable `numbers`. Finally we print the recovered list of numbers and see that it's the same list created in `number_writer.py`:

```
[2, 3, 5, 7, 11, 13]
```

This is a simple way to share data between two programs.

Saving and Reading User-Generated Data

Saving data with `json` is useful when you're working with user-generated data, because if you don't store your user's information somehow, you'll lose it when the program stops running. Let's look at an example where we prompt the user for their name the first time they run a program and then remember their name when they run the program again.

Let's start by storing the user's name:

```
remember
_me.py
import json
❶ username = input("What is your name? ")

filename = 'username.json'
with open(filename, 'w') as f:
❷     json.dump(username, f)
❸     print(f"We'll remember you when you come back, {username}!")
```

At ❶ we prompt for a username to store. Next, we use `json.dump()`, passing it a `username` and a file object, to store the `username` in a file ❷. Then we print a message informing the user that we've stored their information ❸:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Now let's write a new program that greets a user whose name has already been stored:

```
greet_user.py
import json

filename = 'username.json'

with open(filename) as f:
❶     username = json.load(f)
❷     print(f"Welcome back, {username}!")
```

At ❶ we use `json.load()` to read the information stored in `username.json` and assign it to the variable `username`. Now that we've recovered the username, we can welcome them back ❷:

Welcome back, Eric!

We need to combine these two programs into one file. When someone runs `remember_me.py`, we want to retrieve their username from memory if possible; therefore, we'll start with a try block that attempts to recover the username. If the file `username.json` doesn't exist, we'll have the except block prompt for a username and store it in `username.json` for next time:

```
remember
_me.py
import json

# Load the username, if it has been stored previously.
# Otherwise, prompt for the username and store it.
filename = 'username.json'
try:
❶    with open(filename) as f:
❷        username = json.load(f)
❸ except FileNotFoundError:
❹    username = input("What is your name? ")
❺    with open(filename, 'w') as f:
        json.dump(username, f)
        print(f"We'll remember you when you come back, {username}!")
else:
    print(f"Welcome back, {username}!")
```

There's no new code here; blocks of code from the last two examples are just combined into one file. At ❶ we try to open the file `username.json`. If this file exists, we read the username back into memory ❷ and print a message welcoming back the user in the else block. If this is the first time the user runs the program, `username.json` won't exist and a `FileNotFoundException` will occur ❸. Python will move on to the except block where we prompt the user to enter their username ❹. We then use `json.dump()` to store the username and print a greeting ❺.

Whichever block executes, the result is a username and an appropriate greeting. If this is the first time the program runs, this is the output:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Otherwise:

Welcome back, Eric!

This is the output you see if the program was already run at least once.

Refactoring

Often, you'll come to a point where your code will work, but you'll recognize that you could improve the code by breaking it up into a series of functions that have specific jobs. This process is called *refactoring*. Refactoring makes your code cleaner, easier to understand, and easier to extend.

We can refactor *remember_me.py* by moving the bulk of its logic into one or more functions. The focus of *remember_me.py* is on greeting the user, so let's move all of our existing code into a function called `greet_user()`:

```
remember
_me.py
import json

def greet_user():
    """Greet the user by name."""
    filename = 'username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
        username = input("What is your name? ")
        with open(filename, 'w') as f:
            json.dump(username, f)
        print(f"We'll remember you when you come back, {username}!")
    else:
        print(f"Welcome back, {username}!")

greet_user()
```

Because we're using a function now, we update the comments with a docstring that reflects how the program currently works ❶. This file is a little cleaner, but the function `greet_user()` is doing more than just greeting the user—it's also retrieving a stored username if one exists and prompting for a new username if one doesn't exist.

Let's refactor `greet_user()` so it's not doing so many different tasks. We'll start by moving the code for retrieving a stored username to a separate function:

```
import json

def get_stored_username():
    """Get stored username if available."""
    filename = 'username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
        return None
    else:
        return username
```

```
def greet_user():
    """Greet the user by name."""
    username = get_stored_username()
❸    if username:
        print(f"Welcome back, {username}!")
    else:
        username = input("What is your name? ")
        filename = 'username.json'
        with open(filename, 'w') as f:
            json.dump(username, f)
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

The new function `get_stored_username()` has a clear purpose, as stated in the docstring at ❶. This function retrieves a stored username and returns the username if it finds one. If the file `username.json` doesn't exist, the function returns `None` ❷. This is good practice: a function should either return the value you're expecting, or it should return `None`. This allows us to perform a simple test with the return value of the function. At ❸ we print a welcome back message to the user if the attempt to retrieve a username was successful, and if it doesn't, we prompt for a new username.

We should factor one more block of code out of `greet_user()`. If the username doesn't exist, we should move the code that prompts for a new username to a function dedicated to that purpose:

```
import json

def get_stored_username():
    """Get stored username if available."""
    --snip--

def get_new_username():
    """Prompt for a new username."""
    username = input("What is your name? ")
    filename = 'username.json'
    with open(filename, 'w') as f:
        json.dump(username, f)
    return username

def greet_user():
    """Greet the user by name."""
    username = get_stored_username()
    if username:
        print(f"Welcome back, {username}!")
    else:
        username = get_new_username()
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

Each function in this final version of *remember_me.py* has a single, clear purpose. We call `greet_user()`, and that function prints an appropriate message: it either welcomes back an existing user or greets a new user. It does this by calling `get_stored_username()`, which is responsible only for retrieving a stored username if one exists. Finally, `greet_user()` calls `get_new_username()` if necessary, which is responsible only for getting a new username and storing it. This compartmentalization of work is an essential part of writing clear code that will be easy to maintain and extend.

TRY IT YOURSELF

10-11. Favorite Number: Write a program that prompts for the user’s favorite number. Use `json.dump()` to store this number in a file. Write a separate program that reads in this value and prints the message, “I know your favorite number! It’s ____.”

10-12. Favorite Number Remembered: Combine the two programs from Exercise 10-11 into one file. If the number is already stored, report the favorite number to the user. If not, prompt for the user’s favorite number and store it in a file. Run the program twice to see that it works.

10-13. Verify User: The final listing for *remember_me.py* assumes either that the user has already entered their username or that the program is running for the first time. We should modify it in case the current user is not the person who last used the program.

Before printing a welcome back message in `greet_user()`, ask the user if this is the correct username. If it’s not, call `get_new_username()` to get the correct username.

Summary

In this chapter, you learned how to work with files. You learned to read an entire file at once and read through a file’s contents one line at a time. You learned to write to a file and append text onto the end of a file. You read about exceptions and how to handle the exceptions you’re likely to see in your programs. Finally, you learned how to store Python data structures so you can save information your users provide, preventing them from having to start over each time they run a program.

In Chapter 11 you’ll learn efficient ways to test your code. This will help you trust that the code you develop is correct, and it will help you identify bugs that are introduced as you continue to build on the programs you’ve written.

11

TESTING YOUR CODE



When you write a function or a class, you can also write tests for that code. Testing proves that your code works as it's supposed to in response to all the input types it's designed to receive. When you write tests, you can be confident that your code will work correctly as more people begin to use your programs. You'll also be able to test new code as you add it to make sure your changes don't break your program's existing behavior. Every programmer makes mistakes, so every programmer must test their code often, catching problems before users encounter them.

In this chapter you'll learn to test your code using tools in Python's `unittest` module. You'll learn to build a test case and check that a set of inputs results in the output you want. You'll see what a passing test looks like and what a failing test looks like, and you'll learn how a failing test can help you improve your code. You'll learn to test functions and classes, and you'll start to understand how many tests to write for a project.

Testing a Function

To learn about testing, we need code to test. Here's a simple function that takes in a first and last name, and returns a neatly formatted full name:

```
name
_function.py
def get_formatted_name(first, last):
    """Generate a neatly formatted full name."""
    full_name = f'{first} {last}'
    return full_name.title()
```

The function `get_formatted_name()` combines the first and last name with a space in between to complete a full name, and then capitalizes and returns the full name. To check that `get_formatted_name()` works, let's make a program that uses this function. The program `names.py` lets users enter a first and last name, and see a neatly formatted full name:

```
names.py
from name_function import get_formatted_name

print("Enter 'q' at any time to quit.")
while True:
    first = input("\nPlease give me a first name: ")
    if first == 'q':
        break
    last = input("Please give me a last name: ")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print(f"\tNeatly formatted name: {formatted_name}.")
```

This program imports `get_formatted_name()` from `name_function.py`. The user can enter a series of first and last names, and see the formatted full names that are generated:

```
Enter 'q' at any time to quit.

Please give me a first name: janis
Please give me a last name: joplin
    Neatly formatted name: Janis Joplin.

Please give me a first name: bob
Please give me a last name: dylan
    Neatly formatted name: Bob Dylan.

Please give me a first name: q
```

We can see that the names generated here are correct. But let's say we want to modify `get_formatted_name()` so it can also handle middle names. As we do so, we want to make sure we don't break the way the function handles names that have only a first and last name. We could test our code by running `names.py` and entering a name like `Janis Joplin` every time we modify `get_formatted_name()`, but that would become tedious. Fortunately,

Python provides an efficient way to automate the testing of a function's output. If we automate the testing of `get_formatted_name()`, we can always be confident that the function will work when given the kinds of names we've written tests for.

Unit Tests and Test Cases

The module `unittest` from the Python standard library provides tools for testing your code. A *unit test* verifies that one specific aspect of a function's behavior is correct. A *test case* is a collection of unit tests that together prove that a function behaves as it's supposed to, within the full range of situations you expect it to handle. A good test case considers all the possible kinds of input a function could receive and includes tests to represent each of these situations. A test case with *full coverage* includes a full range of unit tests covering all the possible ways you can use a function. Achieving full coverage on a large project can be daunting. It's often good enough to write tests for your code's critical behaviors and then aim for full coverage only if the project starts to see widespread use.

A Passing Test

The syntax for setting up a test case takes some getting used to, but once you've set up the test case it's straightforward to add more unit tests for your functions. To write a test case for a function, import the `unittest` module and the function you want to test. Then create a class that inherits from `unittest.TestCase`, and write a series of methods to test different aspects of your function's behavior.

Here's a test case with one method that verifies that the function `get_formatted_name()` works correctly when given a first and last name:

```
test_name
function.py
import unittest
from name_function import get_formatted_name

❶ class NamesTestCase(unittest.TestCase):
    """Tests for 'name_function.py'."""

    def test_first_last_name(self):
        """Do names like 'Janis Joplin' work?"""
❷        formatted_name = get_formatted_name('janis', 'joplin')
❸        self.assertEqual(formatted_name, 'Janis Joplin')

❹ if __name__ == '__main__':
    unittest.main()
```

First, we import `unittest` and the function we want to test, `get_formatted_name()`. At ❶ we create a class called `NamesTestCase`, which will contain a series of unit tests for `get_formatted_name()`. You can name the class anything you want, but it's best to call it something related to the function you're about to test and to use the word *Test* in the class name. This class must inherit from the class `unittest.TestCase` so Python knows how to run the tests you write.

`NamesTestCase` contains a single method that tests one aspect of `get_formatted_name()`. We call this method `test_first_last_name()` because we're verifying that names with only a first and last name are formatted correctly. Any method that starts with `test_` will be run automatically when we run `test_name_function.py`. Within this test method, we call the function we want to test. In this example we call `get_formatted_name()` with the arguments '`janis`' and '`joplin`', and assign the result to `formatted_name` ❷.

At ❸ we use one of `unittest`'s most useful features: an `assert` method. Assert methods verify that a result you received matches the result you expected to receive. In this case, because we know `get_formatted_name()` is supposed to return a capitalized, properly spaced full name, we expect the value of `formatted_name` to be `Janis Joplin`. To check if this is true, we use `unittest`'s `assertEqual()` method and pass it `formatted_name` and '`Janis Joplin`'. The line

```
self.assertEqual(formatted_name, 'Janis Joplin')
```

says, "Compare the value in `formatted_name` to the string '`Janis Joplin`'. If they are equal as expected, fine. But if they don't match, let me know!"

We're going to run this file directly, but it's important to note that many testing frameworks import your test files before running them. When a file is imported, the interpreter executes the file as it's being imported. The `if` block at ❹ looks at a special variable, `__name__`, which is set when the program is executed. If this file is being run as the main program, the value of `__name__` is set to '`__main__`'. In this case, we call `unittest.main()`, which runs the test case. When a testing framework imports this file, the value of `__name__` won't be '`__main__`' and this block will not be executed.

When we run `test_name_function.py`, we get the following output:

```
.
```

```
Ran 1 test in 0.000s
```

```
OK
```

The dot on the first line of output tells us that a single test passed. The next line tells us that Python ran one test, and it took less than 0.001 seconds to run. The final `OK` tells us that all unit tests in the test case passed.

This output indicates that the function `get_formatted_name()` will always work for names that have a first and last name unless we modify the function. When we modify `get_formatted_name()`, we can run this test again. If the test case passes, we know the function will still work for names like Janis Joplin.

A Failing Test

What does a failing test look like? Let's modify `get_formatted_name()` so it can handle middle names, but we'll do so in a way that breaks the function for names with just a first and last name, like Janis Joplin.

Here's a new version of `get_formatted_name()` that requires a middle name argument:

`name
_function.py`

```
def get_formatted_name(first, middle, last):
    """Generate a neatly formatted full name."""
    full_name = f"{first} {middle} {last}"
    return full_name.title()
```

This version should work for people with middle names, but when we test it, we see that we've broken the function for people with just a first and last name. This time, running the file `test_name_function.py` gives this output:

❶ E
=====

❷ ERROR: test_first_last_name (`__main__.NamesTestCase`)

❸ Traceback (most recent call last):
 File "test_name_function.py", line 8, in test_first_last_name
 formatted_name = get_formatted_name('janis', 'joplin')
TypeError: get_formatted_name() missing 1 required positional argument: 'last'

❹ Ran 1 test in 0.000s
❺ FAILED (errors=1)

There's a lot of information here because there's a lot you might need to know when a test fails. The first item in the output is a single E ❶, which tells us one unit test in the test case resulted in an error. Next, we see that `test_first_last_name()` in `NamesTestCase` caused an error ❷. Knowing which test failed is critical when your test case contains many unit tests. At ❸ we see a standard traceback, which reports that the function call `get_formatted_name('janis', 'joplin')` no longer works because it's missing a required positional argument.

We also see that one unit test was run ❹. Finally, we see an additional message that the overall test case failed and that one error occurred when running the test case ❺. This information appears at the end of the output so you see it right away; you don't need to scroll up through a long output listing to find out how many tests failed.

Responding to a Failed Test

What do you do when a test fails? Assuming you're checking the right conditions, a passing test means the function is behaving correctly and a failing test means there's an error in the new code you wrote. So when a test fails, don't change the test. Instead, fix the code that caused the test to fail. Examine the changes you just made to the function, and figure out how those changes broke the desired behavior.

In this case `get_formatted_name()` used to require only two parameters: a first name and a last name. Now it requires a first name, middle name, and

last name. The addition of that mandatory middle name parameter broke the desired behavior of `get_formatted_name()`. The best option here is to make the middle name optional. Once we do, our test for names like `Janis Joplin` should pass again, and we should be able to accept middle names as well. Let's modify `get_formatted_name()` so middle names are optional and then run the test case again. If it passes, we'll move on to making sure the function handles middle names properly.

To make middle names optional, we move the parameter `middle` to the end of the parameter list in the function definition and give it an empty default value. We also add an `if` test that builds the full name properly, depending on whether or not a middle name is provided:

```
name
_function.py
def get_formatted_name(first, last, middle=''):
    """Generate a neatly formatted full name."""
    if middle:
        full_name = f'{first} {middle} {last}'
    else:
        full_name = f'{first} {last}'
    return full_name.title()
```

In this new version of `get_formatted_name()`, the middle name is optional. If a middle name is passed to the function, the full name will contain a first, middle, and last name. Otherwise, the full name will consist of just a first and last name. Now the function should work for both kinds of names. To find out if the function still works for names like `Janis Joplin`, let's run `test_name_function.py` again:

```
.
-----
Ran 1 test in 0.000s
OK
```

The test case passes now. This is ideal; it means the function works for names like `Janis Joplin` again without us having to test the function manually. Fixing our function was easy because the failed test helped us identify the new code that broke existing behavior.

Adding New Tests

Now that we know `get_formatted_name()` works for simple names again, let's write a second test for people who include a middle name. We do this by adding another method to the class `NamesTestCase`:

```
test_name
_function.py
--snip--
class NamesTestCase(unittest.TestCase):
    """Tests for 'name_function.py'."""
    def test_first_last_name(self):
        --snip--
```

```
def test_first_last_middle_name(self):
    """Do names like 'Wolfgang Amadeus Mozart' work?"""
❶    formatted_name = get_formatted_name(
        'wolfgang', 'mozart', 'amadeus')
    self.assertEqual(formatted_name, 'Wolfgang Amadeus Mozart')

if __name__ == '__main__':
    unittest.main()
```

We name this new method `test_first_last_middle_name()`. The method name must start with `test_` so the method runs automatically when we run `test_name_function.py`. We name the method to make it clear which behavior of `get_formatted_name()` we're testing. As a result, if the test fails, we know right away what kinds of names are affected. It's fine to have long method names in your `TestCase` classes. They need to be descriptive so you can make sense of the output when your tests fail, and because Python calls them automatically, you'll never have to write code that calls these methods.

To test the function, we call `get_formatted_name()` with a first, last, and middle name ❶, and then we use `assertEqual()` to check that the returned full name matches the full name (first, middle, and last) that we expect. When we run `test_name_function.py` again, both tests pass:

```
..
-----
Ran 2 tests in 0.000s
OK
```

Great! We now know that the function still works for names like Janis Joplin, and we can be confident that it will work for names like Wolfgang Amadeus Mozart as well.

TRY IT YOURSELF

11-1. City, Country: Write a function that accepts two parameters: a city name and a country name. The function should return a single string of the form *City, Country*, such as `Santiago, Chile`. Store the function in a module called `city_functions.py`.

Create a file called `test_cities.py` that tests the function you just wrote (remember that you need to import `unittest` and the function you want to test). Write a method called `test_city_country()` to verify that calling your function with values such as `'santiago'` and `'chile'` results in the correct string. Run `test_cities.py`, and make sure `test_city_country()` passes.

(continued)

11-2. Population: Modify your function so it requires a third parameter, `population`. It should now return a single string of the form `City, Country - population xxx`, such as `Santiago, Chile - population 5000000`. Run `test_cities.py` again. Make sure `test_city_country()` fails this time.

Modify the function so the `population` parameter is optional. Run `test_cities.py` again, and make sure `test_city_country()` passes again.

Write a second test called `test_city_country_population()` that verifies you can call your function with the values `'santiago'`, `'chile'`, and `'population=5000000'`. Run `test_cities.py` again, and make sure this new test passes.

Testing a Class

In the first part of this chapter, you wrote tests for a single function. Now you'll write tests for a class. You'll use classes in many of your own programs, so it's helpful to be able to prove that your classes work correctly. If you have passing tests for a class you're working on, you can be confident that improvements you make to the class won't accidentally break its current behavior.

A Variety of Assert Methods

Python provides a number of assert methods in the `unittest.TestCase` class. As mentioned earlier, assert methods test whether a condition you believe is true at a specific point in your code is indeed true. If the condition is true as expected, your assumption about how that part of your program behaves is confirmed; you can be confident that no errors exist. If the condition you assume is true is actually not true, Python raises an exception.

Table 11-1 describes six commonly used assert methods. With these methods you can verify that returned values equal or don't equal expected values, that values are `True` or `False`, and that values are in or not in a given list. You can use these methods only in a class that inherits from `unittest.TestCase`, so let's look at how we can use one of these methods in the context of testing an actual class.

Table 11-1: Assert Methods Available from the `unittest` Module

Method	Use
<code>assertEqual(a, b)</code>	Verify that <code>a == b</code>
<code>assertNotEqual(a, b)</code>	Verify that <code>a != b</code>
<code>assertTrue(x)</code>	Verify that <code>x</code> is <code>True</code>
<code>assertFalse(x)</code>	Verify that <code>x</code> is <code>False</code>
<code>assertIn(item, list)</code>	Verify that <code>item</code> is in <code>list</code>
<code>assertNotIn(item, list)</code>	Verify that <code>item</code> is not in <code>list</code>

A Class to Test

Testing a class is similar to testing a function—much of your work involves testing the behavior of the methods in the class. But there are a few differences, so let's write a class to test. Consider a class that helps administer anonymous surveys:

```
survey.py    class AnonymousSurvey:  
                """Collect anonymous answers to a survey question."""  
  
   ❶    def __init__(self, question):  
            """Store a question, and prepare to store responses."""  
            self.question = question  
            self.responses = []  
  
   ❷    def show_question(self):  
            """Show the survey question."""  
            print(self.question)  
  
   ❸    def store_response(self, new_response):  
            """Store a single response to the survey."""  
            self.responses.append(new_response)  
  
   ❹    def show_results(self):  
            """Show all the responses that have been given."""  
            print("Survey results:")  
            for response in self.responses:  
                print(f"- {response}")
```

This class starts with a survey question that you provide ❶ and includes an empty list to store responses. The class has methods to print the survey question ❷, add a new response to the response list ❸, and print all the responses stored in the list ❹. To create an instance from this class, all you have to provide is a question. Once you have an instance representing a particular survey, you display the survey question with `show_question()`, store a response using `store_response()`, and show results with `show_results()`.

To show that the `AnonymousSurvey` class works, let's write a program that uses the class:

```
language  
_survey.py  from survey import AnonymousSurvey  
  
# Define a question, and make a survey.  
question = "What language did you first learn to speak?"  
my_survey = AnonymousSurvey(question)  
  
# Show the question, and store responses to the question.  
my_survey.show_question()  
print("Enter 'q' at any time to quit.\n")  
while True:  
    response = input("Language: ")  
    if response == 'q':  
        break  
    my_survey.store_response(response)
```

```
# Show the survey results.  
print("\nThank you to everyone who participated in the survey!")  
my_survey.show_results()
```

This program defines a question ("What language did you first learn to speak?") and creates an `AnonymousSurvey` object with that question. The program calls `show_question()` to display the question and then prompts for responses. Each response is stored as it is received. When all responses have been entered (the user inputs `q` to quit), `show_results()` prints the survey results:

```
What language did you first learn to speak?  
Enter 'q' at any time to quit.
```

```
Language: English  
Language: Spanish  
Language: English  
Language: Mandarin  
Language: q
```

```
Thank you to everyone who participated in the survey!  
Survey results:  
- English  
- Spanish  
- English  
- Mandarin
```

This class works for a simple anonymous survey. But let's say we want to improve `AnonymousSurvey` and the module it's in, `survey`. We could allow each user to enter more than one response. We could write a method to list only unique responses and to report how many times each response was given. We could write another class to manage nonanonymous surveys.

Implementing such changes would risk affecting the current behavior of the class `AnonymousSurvey`. For example, it's possible that while trying to allow each user to enter multiple responses, we could accidentally change how single responses are handled. To ensure we don't break existing behavior as we develop this module, we can write tests for the class.

Testing the `AnonymousSurvey` Class

Let's write a test that verifies one aspect of the way `AnonymousSurvey` behaves. We'll write a test to verify that a single response to the survey question is stored properly. We'll use the `assertIn()` method to verify that the response is in the list of responses after it's been stored:

```
test  
_survey.py  
import unittest  
from survey import AnonymousSurvey  
  
❶ class TestAnonymousSurvey(unittest.TestCase):  
    """Tests for the class AnonymousSurvey"""
```

```
❷     def test_store_single_response(self):
        """Test that a single response is stored properly."""
        question = "What language did you first learn to speak?"
❸     my_survey = AnonymousSurvey(question)
        my_survey.store_response('English')
❹     self.assertIn('English', my_survey.responses)

if __name__ == '__main__':
    unittest.main()
```

We start by importing the `unittest` module and the class we want to test, `AnonymousSurvey`. We call our test case `TestAnonymousSurvey`, which again inherits from `unittest.TestCase` ❶. The first test method will verify that when we store a response to the survey question, the response ends up in the survey's list of responses. A good descriptive name for this method is `test_store_single_response()` ❷. If this test fails, we'll know from the method name shown in the output of the failing test that there was a problem storing a single response to the survey.

To test the behavior of a class, we need to make an instance of the class. At ❸ we create an instance called `my_survey` with the question "What language did you first learn to speak?" We store a single response, `English`, using the `store_response()` method. Then we verify that the response was stored correctly by asserting that `English` is in the list `my_survey.responses` ❹.

When we run `test_survey.py`, the test passes:

```
.
```

```
Ran 1 test in 0.001s
```

```
OK
```

This is good, but a survey is useful only if it generates more than one response. Let's verify that three responses can be stored correctly. To do this, we add another method to `TestAnonymousSurvey`:

```
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey"""

    def test_store_single_response(self):
        --snip--

    def test_store_three_responses(self):
        """Test that three individual responses are stored properly."""
        question = "What language did you first learn to speak?"
        my_survey = AnonymousSurvey(question)
❶        responses = ['English', 'Spanish', 'Mandarin']
        for response in responses:
            my_survey.store_response(response)

❷        for response in responses:
```

```
    self.assertIn(response, my_survey.responses)

if __name__ == '__main__':
    unittest.main()
```

We call the new method `test_store_three_responses()`. We create a survey object just like we did in `test_store_single_response()`. We define a list containing three different responses ❶, and then we call `store_response()` for each of these responses. Once the responses have been stored, we write another loop and assert that each response is now in `my_survey.responses` ❷.

When we run `test_survey.py` again, both tests (for a single response and for three responses) pass:

```
..
-----
Ran 2 tests in 0.000s

OK
```

This works perfectly. However, these tests are a bit repetitive, so we'll use another feature of `unittest` to make them more efficient.

The `setUp()` Method

In `test_survey.py` we created a new instance of `AnonymousSurvey` in each test method, and we created new responses in each method. The `unittest.TestCase` class has a `setUp()` method that allows you to create these objects once and then use them in each of your test methods. When you include a `setUp()` method in a `TestCase` class, Python runs the `setUp()` method before running each method starting with `test_`. Any objects created in the `setUp()` method are then available in each test method you write.

Let's use `setUp()` to create a survey instance and a set of responses that can be used in `test_store_single_response()` and `test_store_three_responses()`:

```
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey."""
    def setUp(self):
        """
        Create a survey and a set of responses for use in all test methods.
        """
        question = "What language did you first learn to speak?"
        self.my_survey = AnonymousSurvey(question)
        self.responses = ['English', 'Spanish', 'Mandarin']

    def test_store_single_response(self):
        """Test that a single response is stored properly."""
        self.my_survey.store_response(self.responses[0])
        self.assertIn(self.responses[0], self.my_survey.responses)
```

```
def test_store_three_responses(self):
    """Test that three individual responses are stored properly."""
    for response in self.responses:
        self.my_survey.store_response(response)
    for response in self.responses:
        self.assertIn(response, self.my_survey.responses)

if __name__ == '__main__':
    unittest.main()
```

The method `setUp()` does two things: it creates a survey instance ❶, and it creates a list of responses ❷. Each of these is prefixed by `self`, so they can be used anywhere in the class. This makes the two test methods simpler, because neither one has to make a survey instance or a response. The method `test_store_single_response()` verifies that the first response in `self.responses`—`self.responses[0]`—can be stored correctly, and `test_store_three_responses()` verifies that all three responses in `self.responses` can be stored correctly.

When we run `test_survey.py` again, both tests still pass. These tests would be particularly useful when trying to expand `AnonymousSurvey` to handle multiple responses for each person. After modifying the code to accept multiple responses, you could run these tests and make sure you haven't affected the ability to store a single response or a series of individual responses.

When you're testing your own classes, the `setUp()` method can make your test methods easier to write. You make one set of instances and attributes in `setUp()` and then use these instances in all your test methods. This is much easier than making a new set of instances and attributes in each test method.

NOTE

When a test case is running, Python prints one character for each unit test as it is completed. A passing test prints a dot, a test that results in an error prints an E, and a test that results in a failed assertion prints an F. This is why you'll see a different number of dots and characters on the first line of output when you run your test cases. If a test case takes a long time to run because it contains many unit tests, you can watch these results to get a sense of how many tests are passing.

TRY IT YOURSELF

11-3. Employee: Write a class called `Employee`. The `__init__()` method should take in a first name, a last name, and an annual salary, and store each of these as attributes. Write a method called `give_raise()` that adds \$5,000 to the annual salary by default but also accepts a different raise amount.

Write a test case for `Employee`. Write two test methods, `test_give_default_raise()` and `test_give_custom_raise()`. Use the `setUp()` method so you don't have to create a new `employee` instance in each test method. Run your test case, and make sure both tests pass.

Summary

In this chapter you learned to write tests for functions and classes using tools in the `unittest` module. You learned to write a class that inherits from `unittest.TestCase`, and you learned to write test methods that verify specific behaviors your functions and classes should exhibit. You learned to use the `setUp()` method to efficiently create instances and attributes from your classes that can be used in all the test methods for a class.

Testing is an important topic that many beginners don't learn. You don't have to write tests for all the simple projects you try as a beginner. But as soon as you start to work on projects that involve significant development effort, you should test the critical behaviors of your functions and classes. You'll be more confident that new work on your project won't break the parts that work, and this will give you the freedom to make improvements to your code. If you accidentally break existing functionality, you'll know right away, so you can still fix the problem easily. Responding to a failed test that you ran is much easier than responding to a bug report from an unhappy user.

Other programmers respect your projects more if you include some initial tests. They'll feel more comfortable experimenting with your code and be more willing to work with you on projects. If you want to contribute to a project that other programmers are working on, you'll be expected to show that your code passes existing tests and you'll usually be expected to write tests for new behavior you introduce to the project.

Play around with tests to become familiar with the process of testing your code. Write tests for the most critical behaviors of your functions and classes, but don't aim for full coverage in early projects unless you have a specific reason to do so.

PART II

PROJECTS

Congratulations! You now know enough about Python to start building interactive and meaningful projects. Creating your own projects will teach you new skills and solidify your understanding of the concepts introduced in Part I.

Part II contains three types of projects, and you can choose to do any or all of these projects in whichever order you like. Here's a brief description of each project to help you decide which to dig into first.

Alien Invasion: Making a Game with Python

In the *Alien Invasion* project (Chapters 12, 13, and 14), you'll use the Pygame package to develop a 2D game in which the aim is to shoot down a fleet of aliens as they drop down the screen in levels that increase in speed and difficulty. At the end of the project, you'll have learned skills that will enable you to develop your own 2D games in Pygame.

Data Visualization

The Data Visualization project starts in Chapter 15, in which you'll learn to generate data and create a series of functional and beautiful visualizations of that data using Matplotlib and Plotly. Chapter 16 teaches you to access data from online sources and feed it into a visualization package to create plots of weather data and a map of global earthquake activity. Finally, Chapter 17 shows you how to write a program to automatically download

and visualize data. Learning to make visualizations allows you to explore the field of data mining, which is a highly sought-after skill in the world today.

Web Applications

In the Web Applications project (Chapters 18, 19, and 20), you'll use the Django package to create a simple web application that allows users to keep a journal about any number of topics they've been learning about. Users will create an account with a username and password, enter a topic, and then make entries about what they're learning. You'll also learn how to deploy your app so anyone in the world can access it.

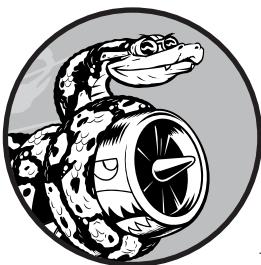
After completing this project, you'll be able to start building your own simple web applications, and you'll be ready to delve into more thorough resources on building applications with Django.

PROJECT 1

ALIEN INVASION

12

A SHIP THAT FIRES BULLETS



Let's build a game called *Alien Invasion*! We'll use Pygame, a collection of fun, powerful Python modules that manage graphics, animation, and even sound, making it easier for you to build sophisticated games. With Pygame handling tasks like drawing images to the screen, you can focus on the higher-level logic of game dynamics.

In this chapter, you'll set up Pygame, and then create a rocket ship that moves right and left and fires bullets in response to player input. In the next two chapters, you'll create a fleet of aliens to destroy, and then continue to refine the game by setting limits on the number of ships you can use and adding a scoreboard.

While building this game, you'll also learn how to manage large projects that span multiple files. We'll refactor a lot of code and manage file contents to organize the project and make the code efficient.

Making games is an ideal way to have fun while learning a language. It's deeply satisfying to play a game you wrote, and writing a simple game will help you comprehend how professionals develop games. As you work through this chapter, enter and run the code to identify how each code block contributes to overall gameplay. Experiment with different values and settings to better understand how to refine interactions in your games.

NOTE

Alien Invasion spans a number of different files, so make a new alien_invasion folder on your system. Be sure to save all files for the project to this folder so your import statements will work correctly.

Also, if you feel comfortable using version control, you might want to use it for this project. If you haven't used version control before, see Appendix D for an overview.

Planning Your Project

When you're building a large project, it's important to prepare a plan before you begin to write code. Your plan will keep you focused and make it more likely that you'll complete the project.

Let's write a description of the general gameplay. Although the following description doesn't cover every detail of *Alien Invasion*, it provides a clear idea of how to start building the game:

In *Alien Invasion*, the player controls a rocket ship that appears at the bottom center of the screen. The player can move the ship right and left using the arrow keys and shoot bullets using the spacebar. When the game begins, a fleet of aliens fills the sky and moves across and down the screen. The player shoots and destroys the aliens. If the player shoots all the aliens, a new fleet appears that moves faster than the previous fleet. If any alien hits the player's ship or reaches the bottom of the screen, the player loses a ship. If the player loses three ships, the game ends.

For the first development phase, we'll make a ship that can move right and left and fires bullets when the player presses the spacebar. After setting up this behavior, we can create the aliens and refine the gameplay.

Installing Pygame

Before you begin coding, install Pygame. The pip module helps you download and install Python packages. To install Pygame, enter the following command at a terminal prompt:

```
$ python -m pip install --user pygame
```

This command tells Python to run the pip module and install the pygame package to the current user's Python installation. If you use a command

other than python to run programs or start a terminal session, such as python3, your command will look like this:

```
$ python3 -m pip install --user pygame
```

NOTE

If this command doesn't work on macOS, try running the command again without the `--user` flag.

Starting the Game Project

We'll begin building the game by creating an empty Pygame window. Later, we'll draw the game elements, such as the ship and the aliens, on this window. We'll also make our game respond to user input, set the background color, and load a ship image.

Creating a Pygame Window and Responding to User Input

We'll make an empty Pygame window by creating a class to represent the game. In your text editor, create a new file and save it as `alien_invasion.py`; then enter the following:

```
alien_invasion.py
import sys

import pygame

class AlienInvasion:
    """Overall class to manage game assets and behavior."""

    def __init__(self):
        """Initialize the game, and create game resources."""
        ❶       pygame.init()

        ❷       self.screen = pygame.display.set_mode((1200, 800))
        pygame.display.set_caption("Alien Invasion")

    def run_game(self):
        """Start the main loop for the game."""
        ❸       while True:
            # Watch for keyboard and mouse events.
            ❹       for event in pygame.event.get():
            ❺       if event.type == pygame.QUIT:
                sys.exit()

            # Make the most recently drawn screen visible.
            ❻       pygame.display.flip()

    if __name__ == '__main__':
        # Make a game instance, and run the game.
        ai = AlienInvasion()
        ai.run_game()
```

First, we import the `sys` and `pygame` modules. The `pygame` module contains the functionality we need to make a game. We'll use tools in the `sys` module to exit the game when the player quits.

Alien Invasion starts as a class called `AlienInvasion`. In the `__init__()` method, the `pygame.init()` function initializes the background settings that Pygame needs to work properly ❶. At ❷, we call `pygame.display.set_mode()` to create a display window, on which we'll draw all the game's graphical elements. The argument `(1200, 800)` is a tuple that defines the dimensions of the game window, which will be 1200 pixels wide by 800 pixels high. (You can adjust these values depending on your display size.) We assign this display window to the attribute `self.screen`, so it will be available in all methods in the class.

The object we assigned to `self.screen` is called a *surface*. A surface in Pygame is a part of the screen where a game element can be displayed. Each element in the game, like an alien or a ship, is its own surface. The surface returned by `display.set_mode()` represents the entire game window. When we activate the game's animation loop, this surface will be redrawn on every pass through the loop, so it can be updated with any changes triggered by user input.

The game is controlled by the `run_game()` method. This method contains a `while` loop ❸ that runs continually. The `while` loop contains an event loop and code that manages screen updates. An *event* is an action that the user performs while playing the game, such as pressing a key or moving the mouse. To make our program respond to events, we write this *event loop* to *listen* for events and perform appropriate tasks depending on the kinds of events that occur. The `for` loop at ❹ is an event loop.

To access the events that Pygame detects, we'll use the `pygame.event.get()` function. This function returns a list of events that have taken place since the last time this function was called. Any keyboard or mouse event will cause this `for` loop to run. Inside the loop, we'll write a series of `if` statements to detect and respond to specific events. For example, when the player clicks the game window's close button, a `pygame.QUIT` event is detected and we call `sys.exit()` to exit the game ❺.

The call to `pygame.display.flip()` at ❻ tells Pygame to make the most recently drawn screen visible. In this case, it simply draws an empty screen on each pass through the `while` loop, erasing the old screen so only the new screen is visible. When we move the game elements around, `pygame.display.flip()` continually updates the display to show the new positions of game elements and hides the old ones, creating the illusion of smooth movement.

At the end of the file, we create an instance of the game, and then call `run_game()`. We place `run_game()` in an `if` block that only runs if the file is called directly. When you run this `alien_invasion.py` file, you should see an empty Pygame window.

Setting the Background Color

Pygame creates a black screen by default, but that's boring. Let's set a different background color. We'll do this at the end of the `__init__()` method.

```

def __init__(self):
    --snip--
    pygame.display.set_caption("Alien Invasion")

    # Set the background color.
❶    self.bg_color = (230, 230, 230)

def run_game(self):
    --snip--
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    # Redraw the screen during each pass through the loop.
❷    self.screen.fill(self.bg_color)

    # Make the most recently drawn screen visible.
    pygame.display.flip()

```

Colors in Pygame are specified as RGB colors: a mix of red, green, and blue. Each color value can range from 0 to 255. The color value (255, 0, 0) is red, (0, 255, 0) is green, and (0, 0, 255) is blue. You can mix different RGB values to create up to 16 million colors. The color value (230, 230, 230) mixes equal amounts of red, blue, and green, which produces a light gray background color. We assign this color to `self.bg_color` ❶.

At ❷, we fill the screen with the background color using the `fill()` method, which acts on a surface and takes only one argument: a color.

Creating a Settings Class

Each time we introduce new functionality into the game, we'll typically create some new settings as well. Instead of adding settings throughout the code, let's write a module called `settings` that contains a class called `Settings` to store all these values in one place. This approach allows us to work with just one settings object any time we need to access an individual setting. This also makes it easier to modify the game's appearance and behavior as our project grows: to modify the game, we'll simply change some values in `settings.py`, which we'll create next, instead of searching for different settings throughout the project.

Create a new file named `settings.py` inside your `alien_invasion` folder, and add this initial `Settings` class:

```

class Settings:
    """A class to store all settings for Alien Invasion."""

    def __init__(self):
        """Initialize the game's settings."""
        # Screen settings
        self.screen_width = 1200
        self.screen_height = 800
        self.bg_color = (230, 230, 230)

```

To make an instance of `Settings` in the project and use it to access our settings, we need to modify `alien_invasion.py` as follows:

```
alien_invasion.py
--snip--
import pygame

from settings import Settings

class AlienInvasion:
    """Overall class to manage game assets and behavior."""

    def __init__(self):
        """Initialize the game, and create game resources."""
        pygame.init()
❶        self.settings = Settings()

❷        self.screen = pygame.display.set_mode(
            (self.settings.screen_width, self.settings.screen_height))
        pygame.display.set_caption("Alien Invasion")

    def run_game(self):
        --snip--
        # Redraw the screen during each pass through the loop.
❸        self.screen.fill(self.settings.bg_color)

        # Make the most recently drawn screen visible.
        pygame.display.flip()
--snip--
```

We import `Settings` into the main program file. Then we create an instance of `Settings` and assign it to `self.settings` ❶, after making the call to `pygame.init()`. When we create a screen ❷, we use the `screen_width` and `screen_height` attributes of `self.settings`, and then we use `self.settings` to access the background color when filling the screen at ❸ as well.

When you run `alien_invasion.py` now you won't yet see any changes, because all we've done is move the settings we were already using elsewhere. Now we're ready to start adding new elements to the screen.

Adding the Ship Image

Let's add the ship to our game. To draw the player's ship on the screen, we'll load an image and then use the Pygame `blit()` method to draw the image.

When you're choosing artwork for your games, be sure to pay attention to licensing. The safest and cheapest way to start is to use freely licensed graphics that you can use and modify, from a website like <https://pixabay.com/>.

You can use almost any type of image file in your game, but it's easiest when you use a bitmap (`.bmp`) file because Pygame loads bitmaps by default. Although you can configure Pygame to use other file types, some file types

depend on certain image libraries that must be installed on your computer. Most images you'll find are in *.jpg* or *.png* formats, but you can convert them to bitmaps using tools like Photoshop, GIMP, and Paint.

Pay particular attention to the background color in your chosen image. Try to find a file with a transparent or solid background that you can replace with any background color using an image editor. Your games will look best if the image's background color matches your game's background color. Alternatively, you can match your game's background to the image's background.

For *Alien Invasion*, you can use the file *ship.bmp* (Figure 12-1), which is available in the book's resources at <https://nostarch.com/pythoncrashcourse2e/>. The file's background color matches the settings we're using in this project. Make a folder called *images* inside your main *alien_invasion* project folder. Save the file *ship.bmp* in the *images* folder.

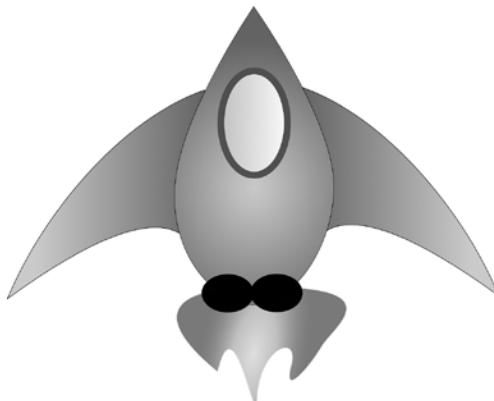


Figure 12-1: The ship for Alien Invasion

Creating the Ship Class

After choosing an image for the ship, we need to display it on the screen. To use our ship, we'll create a new *ship* module that will contain the class *Ship*. This class will manage most of the behavior of the player's ship:

```
ship.py    import pygame

class Ship:
    """A class to manage the ship."""

    def __init__(self, ai_game):
        """Initialize the ship and set its starting position."""
        ❶      self.screen = ai_game.screen
        ❷      self.screen_rect = ai_game.screen.get_rect()

        # Load the ship image and get its rect.
        ❸      self.image = pygame.image.load('images/ship.bmp')
        self.rect = self.image.get_rect()
```

```
# Start each new ship at the bottom center of the screen.  
❸     self.rect.midbottom = self.screen_rect.midbottom  
  
❹     def blitme(self):  
        """Draw the ship at its current location."""  
        self.screen.blit(self.image, self.rect)
```

Pygame is efficient because it lets you treat all game elements like rectangles (*rects*), even if they're not exactly shaped like rectangles. Treating an element as a rectangle is efficient because rectangles are simple geometric shapes. When Pygame needs to figure out whether two game elements have collided, for example, it can do this more quickly if it treats each object as a rectangle. This approach usually works well enough that no one playing the game will notice that we're not working with the exact shape of each game element. We'll treat the ship and the screen as rectangles in this class.

We import the `pygame` module before defining the class. The `__init__()` method of `Ship` takes two parameters: the `self` reference and a reference to the current instance of the `AlienInvasion` class. This will give `Ship` access to all the game resources defined in `AlienInvasion`. At ❶ we assign the screen to an attribute of `Ship`, so we can access it easily in all the methods in this class. At ❷ we access the screen's `rect` attribute using the `get_rect()` method and assign it to `self.screen_rect`. Doing so allows us to place the ship in the correct location on the screen.

To load the image, we call `pygame.image.load()` ❸ and give it the location of our ship image. This function returns a surface representing the ship, which we assign to `self.image`. When the image is loaded, we call `get_rect()` to access the ship surface's `rect` attribute so we can later use it to place the ship.

When you're working with a `rect` object, you can use the x- and y-coordinates of the top, bottom, left, and right edges of the rectangle, as well as the center, to place the object. You can set any of these values to establish the current position of the rect. When you're centering a game element, work with the `center`, `centerx`, or `centery` attributes of a `rect`. When you're working at an edge of the screen, work with the `top`, `bottom`, `left`, or `right` attributes. There are also attributes that combine these properties, such as `midbottom`, `midtop`, `midleft`, and `midright`. When you're adjusting the horizontal or vertical placement of the rect, you can just use the `x` and `y` attributes, which are the x- and y-coordinates of its top-left corner. These attributes spare you from having to do calculations that game developers formerly had to do manually, and you'll use them often.

NOTE

In Pygame, the origin (0, 0) is at the top-left corner of the screen, and coordinates increase as you go down and to the right. On a 1200 by 800 screen, the origin is at the top-left corner, and the bottom-right corner has the coordinates (1200, 800). These coordinates refer to the game window, not the physical screen.

We'll position the ship at the bottom center of the screen. To do so, make the value of `self.rect.midbottom` match the `midbottom` attribute of the screen's rect ❸. Pygame uses these `rect` attributes to position the ship image so it's centered horizontally and aligned with the bottom of the screen.

At ❹, we define the `blitme()` method, which draws the image to the screen at the position specified by `self.rect`.

Drawing the Ship to the Screen

Now let's update `alien_invasion.py` so it creates a ship and calls the ship's `blitme()` method:

`alien_invasion.py`

```
--snip--  
from settings import Settings  
from ship import Ship  
  
class AlienInvasion:  
    """Overall class to manage game assets and behavior."""  
  
    def __init__(self):  
        --snip--  
        pygame.display.set_caption("Alien Invasion")  
  
❶        self.ship = Ship(self)  
  
    def run_game(self):  
        --snip--  
        # Redraw the screen during each pass through the loop.  
        self.screen.fill(self.settings.bg_color)  
❷        self.ship.blitme()  
  
        # Make the most recently drawn screen visible.  
        pygame.display.flip()  
--snip--
```

We import `Ship` and then make an instance of `Ship` after the screen has been created ❶. The call to `Ship()` requires one argument, an instance of `AlienInvasion`. The `self` argument here refers to the current instance of `AlienInvasion`. This is the parameter that gives `Ship` access to the game's resources, such as the `screen` object. We assign this `Ship` instance to `self.ship`.

After filling the background, we draw the ship on the screen by calling `ship.blitme()`, so the ship appears on top of the background ❷.

When you run `alien_invasion.py` now, you should see an empty game screen with the rocket ship sitting at the bottom center, as shown in Figure 12-2.

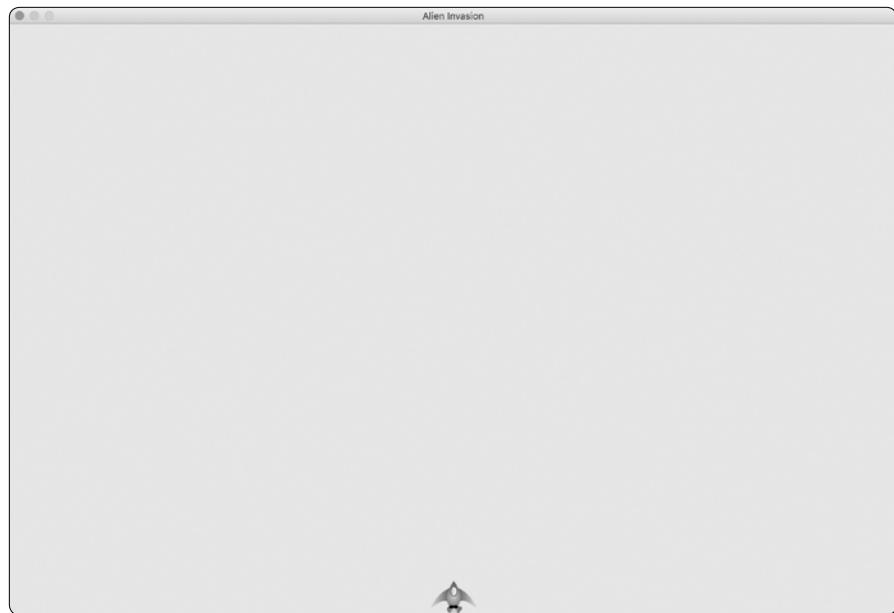


Figure 12-2: Alien Invasion with the ship at the bottom center of the screen

Refactoring: The `_check_events()` and `_update_screen()` Methods

In large projects, you'll often refactor code you've written before adding more code. Refactoring simplifies the structure of the code you've already written, making it easier to build on. In this section, we'll break the `run_game()` method, which is getting lengthy, into two helper methods. A *helper method* does work inside a class but isn't meant to be called through an instance. In Python, a single leading underscore indicates a helper method.

The `_check_events()` Method

We'll move the code that manages events to a separate method called `_check_events()`. This will simplify `run_game()` and isolate the event management loop. Isolating the event loop allows you to manage events separately from other aspects of the game, such as updating the screen.

Here's the `AlienInvasion` class with the new `_check_events()` method, which only affects the code in `run_game()`:

```
alien_invasion.py      def run_game(self):
                        """Start the main loop for the game."""
                        while True:
                            self._check_events()
```

```
# Redraw the screen during each pass through the loop.  
--snip--  
  
❷ def _check_events(self):  
    """Respond to keypresses and mouse events."""  
    for event in pygame.event.get():  
        if event.type == pygame.QUIT:  
            sys.exit()
```

We make a new `_check_events()` method ❷ and move the lines that check whether the player has clicked to close the window into this new method.

To call a method from within a class, use dot notation with the variable `self` and the name of the method ❸. We call the method from inside the `while` loop in `run_game()`.

The `_update_screen()` Method

To further simplify `run_game()`, we'll move the code for updating the screen to a separate method called `_update_screen()`:

alien_invasion.py

```
def run_game(self):  
    """Start the main loop for the game."""  
    while True:  
        self._check_events()  
        self._update_screen()  
  
def _check_events(self):  
    --snip--  
  
def _update_screen(self):  
    """Update images on the screen, and flip to the new screen."""  
    self.screen.fill(self.settings.bg_color)  
    self.ship.blitme()  
  
    pygame.display.flip()
```

We moved the code that draws the background and the ship and flips the screen to `_update_screen()`. Now the body of the main loop in `run_game()` is much simpler. It's easy to see that we're looking for new events and updating the screen on each pass through the loop.

If you've already built a number of games, you'll probably start out by breaking your code into methods like these. But if you've never tackled a project like this, you probably won't know how to structure your code. This approach of writing code that works and then restructuring it as it grows more complex gives you an idea of a realistic development process: you start out writing your code as simply as possible, and then refactor it as your project becomes more complex.

Now that we've restructured the code to make it easier to add to, we can work on the dynamic aspects of the game!

TRY IT YOURSELF

12-1. Blue Sky: Make a Pygame window with a blue background.

12-2. Game Character: Find a bitmap image of a game character you like or convert an image to a bitmap. Make a class that draws the character at the center of the screen and match the background color of the image to the background color of the screen, or vice versa.

Piloting the Ship

Next, we'll give the player the ability to move the ship right and left. We'll write code that responds when the player presses the right or left arrow key. We'll focus on movement to the right first, and then we'll apply the same principles to control movement to the left. As we add this code, you'll learn how to control the movement of images on the screen and respond to user input.

Responding to a Keypress

Whenever the player presses a key, that keypress is registered in Pygame as an event. Each event is picked up by the `pygame.event.get()` method. We need to specify in our `_check_events()` method what kind of events we want the game to check for. Each keypress is registered as a `KEYDOWN` event.

When Pygame detects a `KEYDOWN` event, we need to check whether the key that was pressed is one that triggers a certain action. For example, if the player presses the right arrow key, we want to increase the ship's `rect.x` value to move the ship to the right:

alien_invasion.py

```
def _check_events(self):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
       ❶ elif event.type == pygame.KEYDOWN:
           ❷     if event.key == pygame.K_RIGHT:
                    # Move the ship to the right.
                   ❸     self.ship.rect.x += 1
```

Inside `_check_events()` we add an `elif` block to the event loop to respond when Pygame detects a `KEYDOWN` event ❶. We check whether the key pressed, `event.key`, is the right arrow key ❷. The right arrow key is represented by `pygame.K_RIGHT`. If the right arrow key was pressed, we move the ship to the right by increasing the value of `self.ship.rect.x` by 1 ❸.

When you run *alien_invasion.py* now, the ship should move to the right one pixel every time you press the right arrow key. That's a start, but it's not an efficient way to control the ship. Let's improve this control by allowing continuous movement.

Allowing Continuous Movement

When the player holds down the right arrow key, we want the ship to continue moving right until the player releases the key. We'll have the game detect a `pygame.KEYUP` event so we'll know when the right arrow key is released; then we'll use the `KEYDOWN` and `KEYUP` events together with a flag called `moving_right` to implement continuous motion.

When the `moving_right` flag is `False`, the ship will be motionless. When the player presses the right arrow key, we'll set the flag to `True`, and when the player releases the key, we'll set the flag to `False` again.

The `Ship` class controls all attributes of the ship, so we'll give it an attribute called `moving_right` and an `update()` method to check the status of the `moving_right` flag. The `update()` method will change the position of the ship if the flag is set to `True`. We'll call this method once on each pass through the `while` loop to update the position of the ship.

Here are the changes to `Ship`:

```
ship.py
class Ship:
    """A class to manage the ship."""

    def __init__(self, ai_game):
        --snip--
        # Start each new ship at the bottom center of the screen.
        self.rect.midbottom = self.screen_rect.midbottom

        # Movement flag
❶    self.moving_right = False

❷    def update(self):
        """Update the ship's position based on the movement flag."""
        if self.moving_right:
            self.rect.x += 1

    def blitme(self):
        --snip--
```

We add a `self.moving_right` attribute in the `__init__()` method and set it to `False` initially ❶. Then we add `update()`, which moves the ship right if the flag is `True` ❷. The `update()` method will be called through an instance of `Ship`, so it's not considered a helper method.

Now we need to modify `_check_events()` so that `moving_right` is set to `True` when the right arrow key is pressed and `False` when the key is released:

```
alien_invasion.py
def _check_events(self):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        --snip--
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
                self.ship.moving_right = True
❷            elif event.type == pygame.KEYUP:
```

```
if event.key == pygame.K_RIGHT:  
    self.ship.moving_right = False
```

At ❶, we modify how the game responds when the player presses the right arrow key: instead of changing the ship's position directly, we merely set `moving_right` to `True`. At ❷, we add a new `elif` block, which responds to `KEYUP` events. When the player releases the right arrow key (`K_RIGHT`), we set `moving_right` to `False`.

Next, we modify the `while` loop in `run_game()` so it calls the ship's `update()` method on each pass through the loop:

alien_invasion.py

```
def run_game(self):  
    """Start the main loop for the game."""  
    while True:  
        self._check_events()  
        self.ship.update()  
        self._update_screen()
```

The ship's position will be updated after we've checked for keyboard events and before we update the screen. This allows the ship's position to be updated in response to player input and ensures the updated position will be used when drawing the ship to the screen.

When you run `alien_invasion.py` and hold down the right arrow key, the ship should move continuously to the right until you release the key.

Moving Both Left and Right

Now that the ship can move continuously to the right, adding movement to the left is straightforward. Again, we'll modify the `Ship` class and the `_check_events()` method. Here are the relevant changes to `__init__()` and `update()` in `Ship`:

```
ship.py  
def __init__(self, ai_game):  
    --snip--  
    # Movement flags  
    self.moving_right = False  
    self.moving_left = False  
  
def update(self):  
    """Update the ship's position based on movement flags."""  
    if self.moving_right:  
        self.rect.x += 1  
    if self.moving_left:  
        self.rect.x -= 1
```

In `__init__()`, we add a `self.moving_left` flag. In `update()`, we use two separate `if` blocks rather than an `elif` to allow the ship's `rect.x` value to be increased and then decreased when both arrow keys are held down. This results in the ship standing still. If we used `elif` for motion to the left, the

right arrow key would always have priority. Doing it this way makes the movements more accurate when switching from right to left when the player might momentarily hold down both keys.

We have to make two adjustments to `_check_events()`:

alien_invasion.py

```
def _check_events(self):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        --snip--
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
                self.ship.moving_right = True
            elif event.key == pygame.K_LEFT:
                self.ship.moving_left = True

        elif event.type == pygame.KEYUP:
            if event.key == pygame.K_RIGHT:
                self.ship.moving_right = False
            elif event.key == pygame.K_LEFT:
                self.ship.moving_left = False
```

If a KEYDOWN event occurs for the K_LEFT key, we set `moving_left` to True. If a KEYUP event occurs for the K_LEFT key, we set `moving_left` to False. We can use `elif` blocks here because each event is connected to only one key. If the player presses both keys at once, two separate events will be detected.

When you run *alien_invasion.py* now, you should be able to move the ship continuously to the right and left. If you hold down both keys, the ship should stop moving.

Next, we'll further refine the ship's movement. Let's adjust the ship's speed and limit how far the ship can move so it can't disappear off the sides of the screen.

Adjusting the Ship's Speed

Currently, the ship moves one pixel per cycle through the `while` loop, but we can take finer control of the ship's speed by adding a `ship_speed` attribute to the `Settings` class. We'll use this attribute to determine how far to move the ship on each pass through the loop. Here's the new attribute in *settings.py*:

settings.py

```
class Settings:
    """A class to store all settings for Alien Invasion."""

    def __init__(self):
        --snip--

        # Ship settings
        self.ship_speed = 1.5
```

We set the initial value of `ship_speed` to 1.5. When the ship moves now, its position is adjusted by 1.5 pixels rather than 1 pixel on each pass through the loop.

We're using decimal values for the speed setting to give us finer control of the ship's speed when we increase the tempo of the game later on. However, rect attributes such as `x` store only integer values, so we need to make some modifications to `Ship`:

```
ship.py    class Ship:
              """A class to manage the ship."""

❶    def __init__(self, ai_game):
              """Initialize the ship and set its starting position."""
              self.screen = ai_game.screen
              self.settings = ai_game.settings
              --snip--

              # Start each new ship at the bottom center of the screen.
              --snip--

              # Store a decimal value for the ship's horizontal position.
❷        self.x = float(self.rect.x)

              # Movement flags
              self.moving_right = False
              self.moving_left = False

              def update(self):
              """Update the ship's position based on movement flags."""
              # Update the ship's x value, not the rect.
              if self.moving_right:
❸                  self.x += self.settings.ship_speed
              if self.moving_left:
                  self.x -= self.settings.ship_speed

              # Update rect object from self.x.
❹        self.rect.x = self.x

              def blitme(self):
              --snip--
```

We create a `settings` attribute for `Ship`, so we can use it in `update()` ❶. Because we're adjusting the position of the ship by fractions of a pixel, we need to assign the position to a variable that can store a decimal value. You can use a decimal value to set an attribute of `rect`, but the `rect` will only keep the integer portion of that value. To keep track of the ship's position accurately, we define a new `self.x` attribute that can hold decimal values ❷. We use the `float()` function to convert the value of `self.rect.x` to a decimal and assign this value to `self.x`.

Now when we change the ship's position in `update()`, the value of `self.x` is adjusted by the amount stored in `settings.ship_speed` ❸. After `self.x` has been updated, we use the new value to update `self.rect.x`, which controls

the position of the ship ④. Only the integer portion of `self.x` will be stored in `self.rect.x`, but that's fine for displaying the ship.

Now we can change the value of `ship_speed`, and any value greater than one will make the ship move faster. This will help make the ship respond quickly enough to shoot down aliens, and it will let us change the tempo of the game as the player progresses in gameplay.

NOTE

If you're using macOS, you might notice that the ship moves very slowly, even with a high speed setting. You can remedy this problem by running the game in fullscreen mode, which we'll implement shortly.

Limits the Ship's Range

At this point, the ship will disappear off either edge of the screen if you hold down an arrow key long enough. Let's correct this so the ship stops moving when it reaches the screen's edge. We do this by modifying the `update()` method in `Ship`:

`ship.py`

```
def update(self):
    """Update the ship's position based on movement flags."""
    # Update the ship's x value, not the rect.
❶    if self.moving_right and self.rect.right < self.screen_rect.right:
        self.x += self.settings.ship_speed
❷    if self.moving_left and self.rect.left > 0:
        self.x -= self.settings.ship_speed

    # Update rect object from self.x.
    self.rect.x = self.x
```

This code checks the position of the ship before changing the value of `self.x`. The code `self.rect.right` returns the x-coordinate of the right edge of the ship's rect. If this value is less than the value returned by `self.screen_rect.right`, the ship hasn't reached the right edge of the screen ❶. The same goes for the left edge: if the value of the left side of the rect is greater than zero, the ship hasn't reached the left edge of the screen ❷. This ensures the ship is within these bounds before adjusting the value of `self.x`.

When you run `alien_invasion.py` now, the ship should stop moving at either edge of the screen. This is pretty cool; all we've done is add a conditional test in an `if` statement, but it feels like the ship hits a wall or a force field at either edge of the screen!

Refactoring `_check_events()`

The `_check_events()` method will increase in length as we continue to develop the game, so let's break `_check_events()` into two more methods: one that handles KEYDOWN events and another that handles KEYUP events:

`alien_invasion.py`

```
def _check_events(self):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
```

```
if event.type == pygame.QUIT:
    sys.exit()
elif event.type == pygame.KEYDOWN:
    self._check_keydown_events(event)
elif event.type == pygame.KEYUP:
    self._check_keyup_events(event)

def _check_keydown_events(self, event):
    """Respond to keypresses."""
    if event.key == pygame.K_RIGHT:
        self.ship.moving_right = True
    elif event.key == pygame.K_LEFT:
        self.ship.moving_left = True

def _check_keyup_events(self, event):
    """Respond to key releases."""
    if event.key == pygame.K_RIGHT:
        self.ship.moving_right = False
    elif event.key == pygame.K_LEFT:
        self.ship.moving_left = False
```

We make two new helper methods: `_check_keydown_events()` and `_check_keyup_events()`. Each needs a `self` parameter and an `event` parameter. The bodies of these two methods are copied from `_check_events()`, and we've replaced the old code with calls to the new methods. The `_check_events()` method is simpler now with this cleaner code structure, which will make it easier to develop further responses to player input.

Pressing Q to Quit

Now that we're responding to keypresses efficiently, we can add another way to quit the game. It gets tedious to click the X at the top of the game window to end the game every time you test a new feature, so we'll add a keyboard shortcut to end the game when the player presses Q:

alien_invasion.py

```
def _check_keydown_events(self, event):
    --snip--
    elif event.key == pygame.K_LEFT:
        self.ship.moving_left = True
    elif event.key == pygame.K_q:
        sys.exit()
```

In `_check_keydown_events()`, we add a new block that ends the game when the player presses Q. Now, when testing, you can press Q to close the game rather than using your cursor to close the window.

Running the Game in Fullscreen Mode

Pygame has a fullscreen mode that you might like better than running the game in a regular window. Some games look better in fullscreen mode, and macOS users might see better performance in fullscreen mode.

To run the game in fullscreen mode, make the following changes in `_init_()`:

`alien_invasion.py`

```
def __init__(self):
    """Initialize the game, and create game resources."""
    pygame.init()
    self.settings = Settings()

❶    self.screen = pygame.display.set_mode((0, 0), pygame.FULLSCREEN)
❷    self.settings.screen_width = self.screen.get_rect().width
    self.settings.screen_height = self.screen.get_rect().height
    pygame.display.set_caption("Alien Invasion")
```

When creating the screen surface, we pass a size of `(0, 0)` and the parameter `pygame.FULLSCREEN` ❶. This tells Pygame to figure out a window size that will fill the screen. Because we don't know the width and height of the screen ahead of time, we update these settings after the screen is created ❷. We use the `width` and `height` attributes of the screen's rect to update the `settings` object.

If you like how the game looks or behaves in fullscreen mode, keep these settings. If you liked the game better in its own window, you can revert back to the original approach where we set a specific screen size for the game.

NOTE

Make sure you can quit by pressing Q before running the game in fullscreen mode; Pygame offers no default way to quit a game while in fullscreen mode.

A Quick Recap

In the next section, we'll add the ability to shoot bullets, which involves adding a new file called `bullet.py` and making some modifications to some of the files we're already using. Right now, we have three files containing a number of classes and methods. To be clear about how the project is organized, let's review each of these files before adding more functionality.

`alien_invasion.py`

The main file, `alien_invasion.py`, contains the `AlienInvasion` class. This class creates a number of important attributes used throughout the game: the settings are assigned to `settings`, the main display surface is assigned to `screen`, and a `ship` instance is created in this file as well. The main loop of the game, a `while` loop, is also stored in this module. The `while` loop calls `_check_events()`, `ship.update()`, and `_update_screen()`.

The `_check_events()` method detects relevant events, such as key-presses and releases, and processes each of these types of events through the methods `_check_keydown_events()` and `_check_keyup_events()`. For now,

these methods manage the ship’s movement. The `AlienInvasion` class also contains `_update_screen()`, which redraws the screen on each pass through the main loop.

The `alien_invasion.py` file is the only file you need to run when you want to play *Alien Invasion*. The other files—`settings.py` and `ship.py`—contain code that is imported into this file.

`settings.py`

The `settings.py` file contains the `Settings` class. This class only has an `__init__()` method, which initializes attributes controlling the game’s appearance and the ship’s speed.

`ship.py`

The `ship.py` file contains the `Ship` class. The `Ship` class has an `__init__()` method, an `update()` method to manage the ship’s position, and a `blitme()` method to draw the ship to the screen. The image of the ship is stored in `ship.bmp`, which is in the `images` folder.

TRY IT YOURSELF

12-3. Pygame Documentation: We’re far enough into the game now that you might want to look at some of the Pygame documentation. The Pygame home page is at <https://www.pygame.org/>, and the home page for the documentation is at <https://www.pygame.org/docs/>. Just skim the documentation for now. You won’t need it to complete this project, but it will help if you want to modify *Alien Invasion* or make your own game afterward.

12-4. Rocket: Make a game that begins with a rocket in the center of the screen. Allow the player to move the rocket up, down, left, or right using the four arrow keys. Make sure the rocket never moves beyond any edge of the screen.

12-5. Keys: Make a Pygame file that creates an empty screen. In the event loop, print the `event.key` attribute whenever a `pygame.KEYDOWN` event is detected. Run the program and press various keys to see how Pygame responds.

Shooting Bullets

Now let’s add the ability to shoot bullets. We’ll write code that fires a bullet, which is represented by a small rectangle, when the player presses the spacebar. Bullets will then travel straight up the screen until they disappear off the top of the screen.

Adding the Bullet Settings

At the end of the `__init__()` method, we'll update `settings.py` to include the values we'll need for a new Bullet class:

```
settings.py
def __init__(self):
    --snip--
    # Bullet settings
    self.bullet_speed = 1.0
    self.bullet_width = 3
    self.bullet_height = 15
    self.bullet_color = (60, 60, 60)
```

These settings create dark gray bullets with a width of 3 pixels and a height of 15 pixels. The bullets will travel slightly slower than the ship.

Creating the Bullet Class

Now create a `bullet.py` file to store our Bullet class. Here's the first part of `bullet.py`:

```
bullet.py
import pygame
from pygame.sprite import Sprite

class Bullet(Sprite):
    """A class to manage bullets fired from the ship"""

    def __init__(self, ai_game):
        """Create a bullet object at the ship's current position."""
        super().__init__()
        self.screen = ai_game.screen
        self.settings = ai_game.settings
        self.color = self.settings.bullet_color

        # Create a bullet rect at (0, 0) and then set correct position.
❶      self.rect = pygame.Rect(0, 0, self.settings.bullet_width,
                               self.settings.bullet_height)
❷      self.rect.midtop = ai_game.ship.rect.midtop

        # Store the bullet's position as a decimal value.
❸      self.y = float(self.rect.y)
```

The Bullet class inherits from `Sprite`, which we import from the `pygame.sprite` module. When you use sprites, you can group related elements in your game and act on all the grouped elements at once. To create a bullet instance, `__init__()` needs the current instance of `AlienInvasion`, and we call `super()` to inherit properly from `Sprite`. We also set attributes for the screen and settings objects, and for the bullet's color.

At ❶, we create the bullet's `rect` attribute. The bullet isn't based on an image, so we have to build a rect from scratch using the `pygame.Rect()` class. This class requires the x- and y-coordinates of the top-left corner of the

rect, and the width and height of the rect. We initialize the rect at (0, 0), but we'll move it to the correct location in the next line, because the bullet's position depends on the ship's position. We get the width and height of the bullet from the values stored in self.settings.

At ❷, we set the bullet's midtop attribute to match the ship's midtop attribute. This will make the bullet emerge from the top of the ship, making it look like the bullet is fired from the ship. We store a decimal value for the bullet's y-coordinate so we can make fine adjustments to the bullet's speed ❸.

Here's the second part of *bullet.py*, update() and draw_bullet():

```
bullet.py      def update(self):
                  """Move the bullet up the screen."""
                  # Update the decimal position of the bullet.
❶          self.y -= self.settings.bullet_speed
                  # Update the rect position.
❷          self.rect.y = self.y

      def draw_bullet(self):
                  """Draw the bullet to the screen."""
❸          pygame.draw.rect(self.screen, self.color, self.rect)
```

The update() method manages the bullet's position. When a bullet is fired, it moves up the screen, which corresponds to a decreasing y-coordinate value. To update the position, we subtract the amount stored in settings.bullet_speed from self.y ❶. We then use the value of self.y to set the value of self.rect.y ❷.

The bullet_speed setting allows us to increase the speed of the bullets as the game progresses or as needed to refine the game's behavior. Once a bullet is fired, we never change the value of its x-coordinate, so it will travel vertically in a straight line even if the ship moves.

When we want to draw a bullet, we call draw_bullet(). The draw.rect() function fills the part of the screen defined by the bullet's rect with the color stored in self.color ❸.

Storing Bullets in a Group

Now that we have a Bullet class and the necessary settings defined, we can write code to fire a bullet each time the player presses the spacebar. We'll create a group in AlienInvasion to store all the live bullets so we can manage the bullets that have already been fired. This group will be an instance of the pygame.sprite.Group class, which behaves like a list with some extra functionality that's helpful when building games. We'll use this group to draw bullets to the screen on each pass through the main loop and to update each bullet's position.

We'll create the group in __init__():

```
alien_invasion.py    def __init__(self):
                      --snip--
                      self.ship = Ship(self)
                      self.bullets = pygame.sprite.Group()
```

Then we need to update the position of the bullets on each pass through the `while` loop:

alien_invasion.py

```
def run_game(self):
    """Start the main loop for the game."""
    while True:
        self._check_events()
        self.ship.update()
❶      self.bullets.update()
        self._update_screen()
```

When you call `update()` on a group ❶, the group automatically calls `update()` for each sprite in the group. The line `self.bullets.update()` calls `bullet.update()` for each bullet we place in the group `bullets`.

Firing Bullets

In `AlienInvasion`, we need to modify `_check_keydown_events()` to fire a bullet when the player presses the spacebar. We don't need to change `_check_keyup_events()` because nothing happens when the spacebar is released. We also need to modify `_update_screen()` to make sure each bullet is drawn to the screen before we call `flip()`.

We know there will be a bit of work to do when we fire a bullet, so let's write a new method, `_fire_bullet()`, to handle this work:

alien_invasion.py

```
--snip--
from ship import Ship
❶ from bullet import Bullet

class AlienInvasion:
--snip--
    def _check_keydown_events(self, event):
--snip--
        elif event.key == pygame.K_q:
            sys.exit()
❷        elif event.key == pygame.K_SPACE:
            self._fire_bullet()

    def _check_keyup_events(self, event):
--snip--

    def _fire_bullet(self):
        """Create a new bullet and add it to the bullets group."""
❸        new_bullet = Bullet(self)
❹        self.bullets.add(new_bullet)

    def _update_screen(self):
        """Update images on the screen, and flip to the new screen."""
        self.screen.fill(self.settings.bg_color)
        self.ship.blitme()
❺        for bullet in self.bullets.sprites():
            bullet.draw_bullet()
```

```
    pygame.display.flip()  
--snip--
```

First, we import `Bullet` ❶. Then we call `_fire_bullet()` when the spacebar is pressed ❷. In `_fire_bullet()`, we make an instance of `Bullet` and call it `new_bullet` ❸. We then add it to the group `bullets` using the `add()` method ❹. The `add()` method is similar to `append()`, but it's a method that's written specifically for Pygame groups.

The `bullets.sprites()` method returns a list of all sprites in the group `bullets`. To draw all fired bullets to the screen, we loop through the sprites in `bullets` and call `draw_bullet()` on each one ❺.

When you run `alien_invasion.py` now, you should be able to move the ship right and left, and fire as many bullets as you want. The bullets travel up the screen and disappear when they reach the top, as shown in Figure 12-3. You can alter the size, color, and speed of the bullets in `settings.py`.

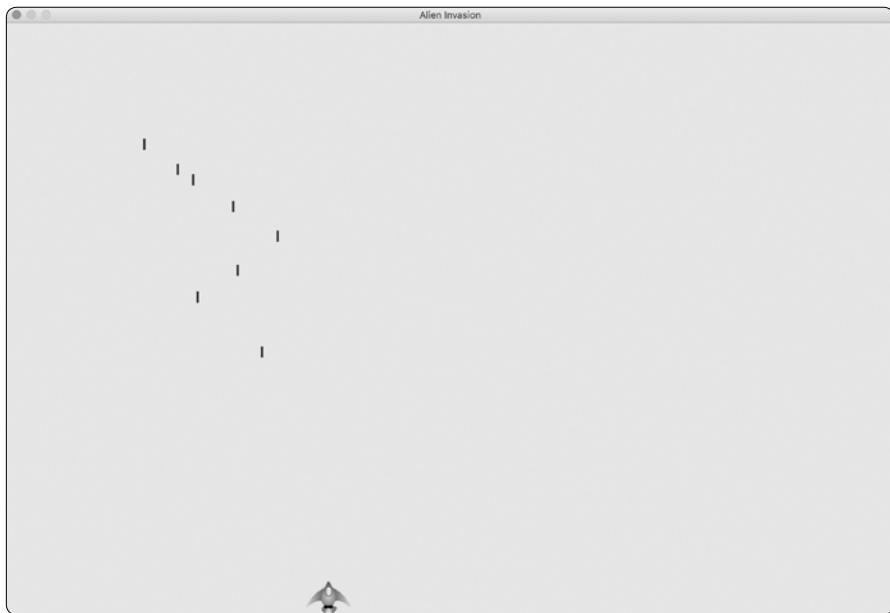


Figure 12-3: The ship after firing a series of bullets

Deleting Old Bullets

At the moment, the bullets disappear when they reach the top, but only because Pygame can't draw them above the top of the screen. The bullets actually continue to exist; their y-coordinate values just grow increasingly negative. This is a problem, because they continue to consume memory and processing power.

We need to get rid of these old bullets, or the game will slow down from doing so much unnecessary work. To do this, we need to detect when the `bottom` value of a bullet’s `rect` has a value of 0, which indicates the bullet has passed off the top of the screen:

alien_invasion.py

```
def run_game(self):
    """Start the main loop for the game."""
    while True:
        self._check_events()
        self.ship.update()
        self.bullets.update()

        # Get rid of bullets that have disappeared.
❶       for bullet in self.bullets.copy():
❷           if bullet.rect.bottom <= 0:
❸               self.bullets.remove(bullet)
❹       print(len(self.bullets))

        self._update_screen()
```

When you use a `for` loop with a list (or a group in Pygame), Python expects that the list will stay the same length as long as the loop is running. Because we can’t remove items from a list or group within a `for` loop, we have to loop over a copy of the group. We use the `copy()` method to set up the `for` loop ❶, which enables us to modify bullets inside the loop. We check each bullet to see whether it has disappeared off the top of the screen at ❷. If it has, we remove it from `bullets` ❸. At ❹ we insert a `print()` call to show how many bullets currently exist in the game and verify that they’re being deleted when they reach the top of the screen.

If this code works correctly, we can watch the terminal output while firing bullets and see that the number of bullets decreases to zero after each series of bullets has cleared the top of the screen. After you run the game and verify that bullets are being deleted properly, remove the `print()` call. If you leave it in, the game will slow down significantly because it takes more time to write output to the terminal than it does to draw graphics to the game window.

Limiting the Number of Bullets

Many shooting games limit the number of bullets a player can have on the screen at one time; doing so encourages players to shoot accurately. We’ll do the same in *Alien Invasion*.

First, store the number of bullets allowed in `settings.py`:

settings.py

```
# Bullet settings
--snip--
self.bullet_color = (60, 60, 60)
self.bullets_allowed = 3
```

This limits the player to three bullets at a time. We'll use this setting in `AlienInvasion` to check how many bullets exist before creating a new bullet in `_fire_bullet()`:

alien_invasion.py

```
def _fire_bullet(self):
    """Create a new bullet and add it to the bullets group."""
    if len(self.bullets) < self.settings.bullets_allowed:
        new_bullet = Bullet(self)
        self.bullets.add(new_bullet)
```

When the player presses the spacebar, we check the length of `bullets`. If `len(self.bullets)` is less than three, we create a new bullet. But if three bullets are already active, nothing happens when the spacebar is pressed. When you run the game now, you should be able to fire bullets only in groups of three.

Creating the `_update_bullets()` Method

We want to keep the `AlienInvasion` class reasonably well organized, so now that we've written and checked the bullet management code, we can move it to a separate method. We'll create a new method called `_update_bullets()` and add it just before `_update_screen()`:

alien_invasion.py

```
def _update_bullets(self):
    """Update position of bullets and get rid of old bullets."""
    # Update bullet positions.
    self.bullets.update()

    # Get rid of bullets that have disappeared.
    for bullet in self.bullets.copy():
        if bullet.rect.bottom <= 0:
            self.bullets.remove(bullet)
```

The code for `_update_bullets()` is cut and pasted from `run_game()`; all we've done here is clarify the comments.

The `while` loop in `run_game()` looks simple again:

alien_invasion.py

```
while True:
    self._check_events()
    self.ship.update()
    self._update_bullets()
    self._update_screen()
```

Now our main loop contains only minimal code, so we can quickly read the method names and understand what's happening in the game. The main loop checks for player input, and then updates the position of the ship and any bullets that have been fired. We then use the updated positions to draw a new screen.

Run `alien_invasion.py` one more time, and make sure you can still fire bullets without errors.

TRY IT YOURSELF

12-6. Sideways Shooter: Write a game that places a ship on the left side of the screen and allows the player to move the ship up and down. Make the ship fire a bullet that travels right across the screen when the player presses the spacebar. Make sure bullets are deleted once they disappear off the screen.

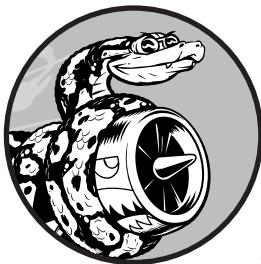
Summary

In this chapter, you learned to make a plan for a game and learned the basic structure of a game written in Pygame. You learned to set a background color and store settings in a separate class where you can adjust them more easily. You saw how to draw an image to the screen and give the player control over the movement of game elements. You created elements that move on their own, like bullets flying up a screen, and deleted objects that are no longer needed. You also learned to refactor code in a project on a regular basis to facilitate ongoing development.

In Chapter 13, we'll add aliens to *Alien Invasion*. By the end of the chapter, you'll be able to shoot down aliens, hopefully before they reach your ship!

13

ALIENS!



In this chapter, we'll add aliens to *Alien Invasion*. We'll add one alien near the top of the screen and then generate a whole fleet of aliens. We'll make the fleet advance sideways and down, and we'll get rid of any aliens hit by a bullet. Finally, we'll limit the number of ships a player has and end the game when the player runs out of ships.

As you work through this chapter, you'll learn more about Pygame and about managing a large project. You'll also learn to detect collisions between game objects, like bullets and aliens. Detecting collisions helps you define interactions between elements in your games: for example, you can

confine a character inside the walls of a maze or pass a ball between two characters. We'll continue to work from a plan that we revisit occasionally to maintain the focus of our code-writing sessions.

Before we start writing new code to add a fleet of aliens to the screen, let's look at the project and update our plan.

Reviewing the Project

When you're beginning a new phase of development on a large project, it's always a good idea to revisit your plan and clarify what you want to accomplish with the code you're about to write. In this chapter, we'll:

- Examine our code and determine if we need to refactor before implementing new features.
- Add a single alien to the top-left corner of the screen with appropriate spacing around it.
- Use the spacing around the first alien and the overall screen size to determine how many aliens can fit on the screen. We'll write a loop to create aliens to fill the upper portion of the screen.
- Make the fleet move sideways and down until the entire fleet is shot down, an alien hits the ship, or an alien reaches the ground. If the entire fleet is shot down, we'll create a new fleet. If an alien hits the ship or the ground, we'll destroy the ship and create a new fleet.
- Limit the number of ships the player can use, and end the game when the player has used up the allotted number of ships.

We'll refine this plan as we implement features, but this is sufficient to start with.

You should also review your existing code when you begin working on a new series of features in a project. Because each new phase typically makes a project more complex, it's best to clean up any cluttered or inefficient code. We've been refactoring as we go, so there isn't any code that we need to refactor at this point.

Creating the First Alien

Placing one alien on the screen is like placing a ship on the screen. Each alien's behavior is controlled by a class called `Alien`, which we'll structure like the `Ship` class. We'll continue using bitmap images for simplicity. You can find your own image for an alien or use the one shown in Figure 13-1, which is available in the book's resources at <https://nostarch.com/pythoncrashcourse2e/>. This image has a gray background, which matches the screen's background color. Make sure you save the image file you choose in the `images` folder.



Figure 13-1: The alien we'll use to build the fleet

Creating the Alien Class

Now we'll write the Alien class and save it as *alien.py*:

```
alien.py import pygame
from pygame.sprite import Sprite

class Alien(Sprite):
    """A class to represent a single alien in the fleet."""

    def __init__(self, ai_game):
        """Initialize the alien and set its starting position."""
        super().__init__()
        self.screen = ai_game.screen

        # Load the alien image and set its rect attribute.
        self.image = pygame.image.load('images/alien.bmp')
        self.rect = self.image.get_rect()

        # Start each new alien near the top left of the screen.
❶      self.rect.x = self.rect.width
❷      self.rect.y = self.rect.height

        # Store the alien's exact horizontal position.
❸      self.x = float(self.rect.x)
```

Most of this class is like the *Ship* class except for the aliens' placement on the screen. We initially place each alien near the top-left corner of the screen; we add a space to the left of it that's equal to the alien's width and a space above it equal to its height **❶** so it's easy to see. We're mainly

concerned with the aliens' horizontal speed, so we'll track the horizontal position of each alien precisely ❷.

This Alien class doesn't need a method for drawing it to the screen; instead, we'll use a Pygame group method that automatically draws all the elements of a group to the screen.

Creating an Instance of the Alien

We want to create an instance of Alien so we can see the first alien on the screen. Because it's part of our setup work, we'll add the code for this instance at the end of the `__init__()` method in `AlienInvasion`. Eventually, we'll create an entire fleet of aliens, which will be quite a bit of work, so we'll make a new helper method called `_create_fleet()`.

The order of methods in a class doesn't matter, as long as there's some consistency to how they're placed. I'll place `_create_fleet()` just before the `_update_screen()` method, but anywhere in `AlienInvasion` will work. First, we'll import the Alien class.

Here are the updated `import` statements for `alien_invasion.py`:

alien_invasion.py

```
--snip--  
from bullet import Bullet  
from alien import Alien
```

And here's the updated `__init__()` method:

alien_invasion.py

```
def __init__(self):  
    --snip--  
    self.ship = Ship(self)  
    self.bullets = pygame.sprite.Group()  
    self.aliens = pygame.sprite.Group()  
  
    self._create_fleet()
```

We create a group to hold the fleet of aliens, and we call `_create_fleet()`, which we're about to write.

Here's the new `_create_fleet()` method:

alien_invasion.py

```
def _create_fleet(self):  
    """Create the fleet of aliens."""  
    # Make an alien.  
    alien = Alien(self)  
    self.aliens.add(alien)
```

In this method, we're creating one instance of Alien, and then adding it to the group that will hold the fleet. The alien will be placed in the default upper-left area of the screen, which is perfect for the first alien.

To make the alien appear, we need to call the group's `draw()` method in `_update_screen()`:

alien_invasion.py

```
def _update_screen(self):
    --snip--
    for bullet in self.bullets.sprites():
        bullet.draw_bullet()
    self.aliens.draw(self.screen)

    pygame.display.flip()
```

When you call `draw()` on a group, Pygame draws each element in the group at the position defined by its `rect` attribute. The `draw()` method requires one argument: a surface on which to draw the elements from the group. Figure 13-2 shows the first alien on the screen.

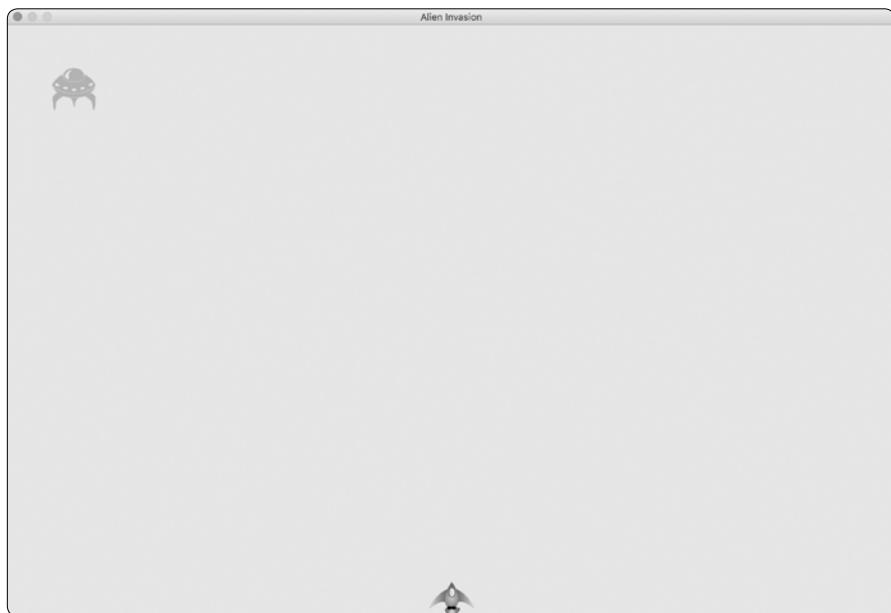


Figure 13-2: The first alien appears.

Now that the first alien appears correctly, we'll write the code to draw an entire fleet.

Building the Alien Fleet

To draw a fleet, we need to figure out how many aliens can fit across the screen and how many rows of aliens can fit down the screen. We'll first figure out the horizontal spacing between aliens and create a row; then we'll determine the vertical spacing and create an entire fleet.

Determining How Many Aliens Fit in a Row

To figure out how many aliens fit in a row, let's look at how much horizontal space we have. The screen width is stored in `settings.screen_width`, but we need an empty margin on either side of the screen. We'll make this margin the width of one alien. Because we have two margins, the available space for aliens is the screen width minus two alien widths:

```
available_space_x = settings.screen_width - (2 * alien_width)
```

We also need to set the spacing between aliens; we'll make it one alien width. The space needed to display one alien is twice its width: one width for the alien and one width for the empty space to its right. To find the number of aliens that fit across the screen, we divide the available space by two times the width of an alien. We use *floor division* (`//`), which divides two numbers and drops any remainder, so we'll get an integer number of aliens:

```
number_aliens_x = available_space_x // (2 * alien_width)
```

We'll use these calculations when we create the fleet.

NOTE

One great aspect of calculations in programming is that you don't have to be sure your formulas are correct when you first write them. You can try them out and see if they work. At worst, you'll have a screen that's overcrowded with aliens or has too few aliens. You can then revise your calculations based on what you see on the screen.

Creating a Row of Aliens

We're ready to generate a full row of aliens. Because our code for making a single alien works, we'll rewrite `_create_fleet()` to make a whole row of aliens:

```
alien_invasion.py    def _create_fleet(self):
                      """Create the fleet of aliens."""
                      # Create an alien and find the number of aliens in a row.
                      # Spacing between each alien is equal to one alien width.
    ❶      alien = Alien(self)
    ❷      alien_width = alien.rect.width
    ❸      available_space_x = self.settings.screen_width - (2 * alien_width)
            number_aliens_x = available_space_x // (2 * alien_width)

                      # Create the first row of aliens.
    ❹      for alien_number in range(number_aliens_x):
                          # Create an alien and place it in the row.
                          alien = Alien(self)
    ❺      alien.x = alien_width + 2 * alien_width * alien_number
                          alien.rect.x = alien.x
                          self.aliens.add(alien)
```

We've already thought through most of this code. We need to know the alien's width and height to place aliens, so we create an alien at ❶ before we perform calculations. This alien won't be part of the fleet, so don't add it to the group `aliens`. At ❷ we get the alien's width from its `rect` attribute and store this value in `alien_width` so we don't have to keep working through the `rect` attribute. At ❸ we calculate the horizontal space available for aliens and the number of aliens that can fit into that space.

Next, we set up a loop that counts from 0 to the number of aliens we need to make ❹. In the main body of the loop, we create a new alien and then set its x-coordinate value to place it in the row ❺. Each alien is pushed to the right one alien width from the left margin. Next, we multiply the alien width by 2 to account for the space each alien takes up, including the empty space to its right, and we multiply this amount by the alien's position in the row. We use the alien's `x` attribute to set the position of its `rect`. Then we add each new alien to the group `aliens`.

When you run *Alien Invasion* now, you should see the first row of aliens appear, as in Figure 13-3.



Figure 13-3: The first row of aliens

The first row is offset to the left, which is actually good for gameplay. The reason is that we want the fleet to move right until it hits the edge of the screen, then drop down a bit, then move left, and so forth. Like the classic game *Space Invaders*, this movement is more interesting than having the fleet drop straight down. We'll continue this motion until all aliens are shot down or until an alien hits the ship or the bottom of the screen.

NOTE

Depending on the screen width you've chosen, the alignment of the first row of aliens might look slightly different on your system.

Refactoring `_create_fleet()`

If the code we've written so far was all we need to create a fleet, we'd probably leave `_create_fleet()` as is. But we have more work to do, so let's clean up the method a bit. We'll add a new helper method, `_create_alien()`, and call it from `_create_fleet()`:

`alien_invasion.py`

```
def _create_fleet(self):
    --snip--
    # Create the first row of aliens.
    for alien_number in range(number_of.aliens_x):
        self._create_alien(alien_number)

def _create_alien(self, alien_number):
    """Create an alien and place it in the row."""
    alien = Alien(self)
    alien_width = alien.rect.width
    alien.x = alien_width + 2 * alien_width * alien_number
    alien.rect.x = alien.x
    self.aliens.add(alien)
```

The method `_create_alien()` requires one parameter in addition to `self`: it needs the alien number that's currently being created. We use the same body we made for `_create_fleet()` except that we get the width of an alien inside the method instead of passing it as an argument. This refactoring will make it easier to add new rows and create an entire fleet.

Adding Rows

To finish the fleet, we'll determine the number of rows that fit on the screen and then repeat the loop for creating the aliens in one row until we have the correct number of rows. To determine the number of rows, we find the available vertical space by subtracting the alien height from the top, the ship height from the bottom, and two alien heights from the bottom of the screen:

```
available_space_y = settings.screen_height - (3 * alien_height) - ship_height
```

The result will create some empty space above the ship, so the player has some time to start shooting aliens at the beginning of each level.

Each row needs some empty space below it, which we'll make equal to the height of one alien. To find the number of rows, we divide the available space by two times the height of an alien. We use floor division because we can only make an integer number of rows. (Again, if these calculations are off, we'll see it right away and adjust our approach until we have reasonable spacing.)

```
number_rows = available_height_y // (2 * alien_height)
```

Now that we know how many rows fit in a fleet, we can repeat the code for creating a row:

alien_invasion.py

```
def _create_fleet(self):
    --snip--
    alien = Alien(self)
❶    alien_width, alien_height = alien.rect.size
    available_space_x = self.settings.screen_width - (2 * alien_width)
    number_aliens_x = available_space_x // (2 * alien_width)

    # Determine the number of rows of aliens that fit on the screen.
    ship_height = self.ship.rect.height
❷    available_space_y = (self.settings.screen_height -
                           (3 * alien_height) - ship_height)
    number_rows = available_space_y // (2 * alien_height)

    # Create the full fleet of aliens.
❸    for row_number in range(number_rows):
        for alien_number in range(number_aliens_x):
            self._create_alien(alien_number, row_number)

def _create_alien(self, alien_number, row_number):
    """Create an alien and place it in the row."""
    alien = Alien(self)
    alien_width, alien_height = alien.rect.size
    alien.x = alien_width + 2 * alien_width * alien_number
    alien.rect.x = alien.x
❹    alien.rect.y = alien.rect.height + 2 * alien.rect.height * row_number
    self.aliens.add(alien)
```

We need the width and height of an alien, so at ❶ we use the attribute `size`, which contains a tuple with the width and height of a `rect` object. To calculate the number of rows we can fit on the screen, we write our `available_space_y` calculation right after the calculation for `available_space_x` ❷. The calculation is wrapped in parentheses so the outcome can be split over two lines, which results in lines of 79 characters or less, as is recommended.

To create multiple rows, we use two nested loops: one outer and one inner loop ❸. The inner loop creates the aliens in one row. The outer loop counts from 0 to the number of rows we want; Python uses the code for making a single row and repeats it `number_rows` times.

To nest the loops, write the new `for` loop and indent the code you want to repeat. (Most text editors make it easy to indent and unindent blocks of code, but for help see Appendix B.) Now when we call `_create_alien()`, we include an argument for the row number so each row can be placed farther down the screen.

The definition of `_create_alien()` needs a parameter to hold the row number. Within `_create_alien()`, we change an alien's y-coordinate value when it's not in the first row ❹ by starting with one alien's height to create empty space at the top of the screen. Each row starts two alien heights below

the previous row, so we multiply the alien height by two and then by the row number. The first row number is 0, so the vertical placement of the first row is unchanged. All subsequent rows are placed farther down the screen.

When you run the game now, you should see a full fleet of aliens, as shown in Figure 13-4.

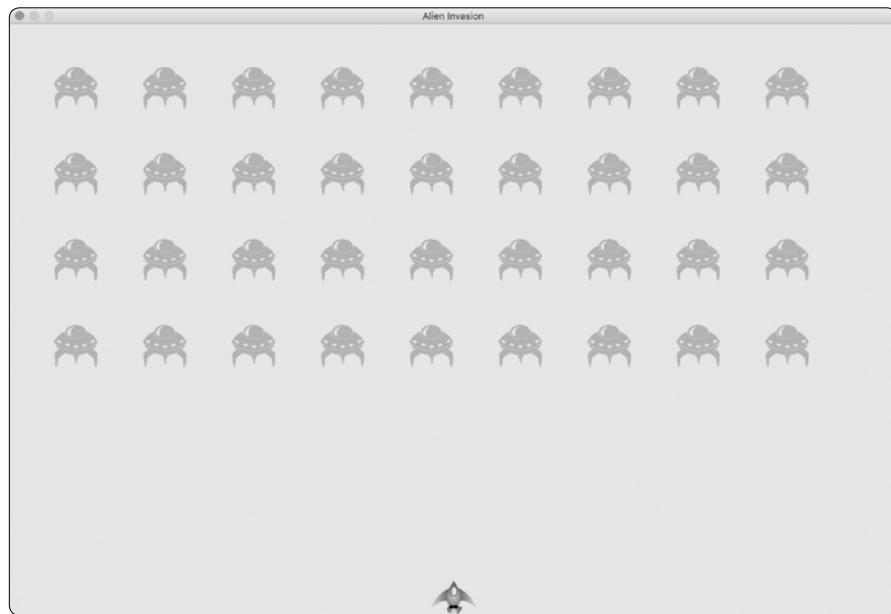


Figure 13-4: The full fleet appears.

In the next section, we'll make the fleet move!

TRY IT YOURSELF

13-1. Stars: Find an image of a star. Make a grid of stars appear on the screen.

13-2. Better Stars: You can make a more realistic star pattern by introducing randomness when you place each star. Recall that you can get a random number like this:

```
from random import randint
random_number = randint(-10, 10)
```

This code returns a random integer between -10 and 10. Using your code in Exercise 13-1, adjust each star's position by a random amount.

Making the Fleet Move

Now let's make the fleet of aliens move to the right across the screen until it hits the edge, and then make it drop a set amount and move in the other direction. We'll continue this movement until all aliens have been shot down, one collides with the ship, or one reaches the bottom of the screen. Let's begin by making the fleet move to the right.

Moving the Aliens Right

To move the aliens, we'll use an `update()` method in `alien.py`, which we'll call for each alien in the group of aliens. First, add a setting to control the speed of each alien:

`settings.py`

```
def __init__(self):
    --snip--
    # Alien settings
    self.alien_speed = 1.0
```

Then use this setting to implement `update()`:

`alien.py`

```
def __init__(self, ai_game):
    """Initialize the alien and set its starting position."""
    super().__init__()
    self.screen = ai_game.screen
    self.settings = ai_game.settings
    --snip--

    def update(self):
        """Move the alien to the right."""
       ❶        self.x += self.settings.alien_speed
       ❷        self.rect.x = self.x
```

We create a `settings` parameter in `__init__()` so we can access the alien's speed in `update()`. Each time we update an alien's position, we move it to the right by the amount stored in `alien_speed`. We track the alien's exact position with the `self.x` attribute, which can hold decimal values ❶. We then use the value of `self.x` to update the position of the alien's rect ❷.

In the main while loop, we already have calls to update the ship and bullet positions. Now we'll add a call to update the position of each alien as well:

`alien_invasion.py`

```
while True:
    self._check_events()
    self.ship.update()
    self._update_bullets()
    self._update.aliens()
    self._update_screen()
```

We're about to write some code to manage the movement of the fleet, so we create a new method called `_update.aliens()`. We set the aliens' positions to update after the bullets have been updated, because we'll soon be checking to see whether any bullets hit any aliens.

Where you place this method in the module is not critical. But to keep the code organized, I'll place it just after `_update.bullets()` to match the order of method calls in the `while` loop. Here's the first version of `_update.aliens()`:

alien_invasion.py

```
def _update.aliens(self):
    """Update the positions of all aliens in the fleet."""
    self.aliens.update()
```

We use the `update()` method on the `aliens` group, which calls each alien's `update()` method. When you run *Alien Invasion* now, you should see the fleet move right and disappear off the side of the screen.

Creating Settings for Fleet Direction

Now we'll create the settings that will make the fleet move down the screen and to the left when it hits the right edge of the screen. Here's how to implement this behavior:

settings.py

```
# Alien settings
self.alien_speed = 1.0
self.fleet_drop_speed = 10
# fleet_direction of 1 represents right; -1 represents left.
self.fleet_direction = 1
```

The setting `fleet_drop_speed` controls how quickly the fleet drops down the screen each time an alien reaches either edge. It's helpful to separate this speed from the aliens' horizontal speed so you can adjust the two speeds independently.

To implement the setting `fleet_direction`, we could use a text value, such as 'left' or 'right', but we'd end up with `if-elif` statements testing for the fleet direction. Instead, because we have only two directions to deal with, let's use the values 1 and -1, and switch between them each time the fleet changes direction. (Using numbers also makes sense because moving right involves adding to each alien's x-coordinate value, and moving left involves subtracting from each alien's x-coordinate value.)

Checking Whether an Alien Has Hit the Edge

We need a method to check whether an alien is at either edge, and we need to modify `update()` to allow each alien to move in the appropriate direction. This code is part of the `Alien` class:

alien.py

```
def check_edges(self):
    """Return True if alien is at edge of screen."""
    screen_rect = self.screen.get_rect()
```

```
❶     if self.rect.right >= screen_rect.right or self.rect.left <= 0:
         return True

    def update(self):
        """Move the alien right or left."""
❷        self.x += (self.settings.alien_speed *
                    self.settings.fleet_direction)
        self.rect.x = self.x
```

We can call the new method `check_edges()` on any alien to see whether it's at the left or right edge. The alien is at the right edge if the `right` attribute of its `rect` is greater than or equal to the `right` attribute of the screen's `rect`. It's at the left edge if its `left` value is less than or equal to 0 ❶.

We modify the method `update()` to allow motion to the left or right by multiplying the alien's speed by the value of `fleet_direction` ❷. If `fleet_direction` is 1, the value of `alien_speed` will be added to the alien's current position, moving the alien to the right; if `fleet_direction` is -1, the value will be subtracted from the alien's position, moving the alien to the left.

Dropping the Fleet and Changing Direction

When an alien reaches the edge, the entire fleet needs to drop down and change direction. Therefore, we need to add some code to `AlienInvasion` because that's where we'll check whether any aliens are at the left or right edge. We'll make this happen by writing the methods `_check_fleet_edges()` and `_change_fleet_direction()`, and then modifying `_update.aliens()`. I'll put these new methods after `_create_alien()`, but again the placement of these methods in the class isn't critical.

```
alien_invasion.py      def _check_fleet_edges(self):
                        """Respond appropriately if any aliens have reached an edge."""
❶        for alien in self.aliens.sprites():
            if alien.check_edges():
❷            self._change_fleet_direction()
            break

    def _change_fleet_direction(self):
        """Drop the entire fleet and change the fleet's direction."""
        for alien in self.aliens.sprites():
❸            alien.rect.y += self.settings.fleet_drop_speed
            self.settings.fleet_direction *= -1
```

In `_check_fleet_edges()`, we loop through the fleet and call `check_edges()` on each alien ❶. If `check_edges()` returns `True`, we know an alien is at an edge and the whole fleet needs to change direction; so we call `_change_fleet_direction()` and break out of the loop ❷. In `_change_fleet_direction()`, we loop through all the aliens and drop each one using the setting `fleet_drop_speed` ❸; then we change the value of `fleet_direction` by multiplying its current value by -1. The line that changes the fleet's direction isn't part of the `for` loop. We want to change each alien's vertical position, but we only want to change the direction of the fleet once.

Here are the changes to `_update.aliens()`:

`alien_invasion.py`

```
def _update.aliens(self):
    """
    Check if the fleet is at an edge,
    then update the positions of all aliens in the fleet.
    """
    self._check_fleet_edges()
    self.aliens.update()
```

We've modified the method by calling `_check_fleet_edges()` before updating each alien's position.

When you run the game now, the fleet should move back and forth between the edges of the screen and drop down every time it hits an edge. Now we can start shooting down aliens and watch for any aliens that hit the ship or reach the bottom of the screen.

TRY IT YOURSELF

13-3. Raindrops: Find an image of a raindrop and create a grid of raindrops. Make the raindrops fall toward the bottom of the screen until they disappear.

13-4. Steady Rain: Modify your code in Exercise 13-3 so when a row of raindrops disappears off the bottom of the screen, a new row appears at the top of the screen and begins to fall.

Shooting Aliens

We've built our ship and a fleet of aliens, but when the bullets reach the aliens, they simply pass through because we aren't checking for collisions. In game programming, *collisions* happen when game elements overlap. To make the bullets shoot down aliens, we'll use the method `sprite.groupcollide()` to look for collisions between members of two groups.

Detecting Bullet Collisions

We want to know right away when a bullet hits an alien so we can make an alien disappear as soon as it's hit. To do this, we'll look for collisions immediately after updating the position of all the bullets.

The `sprite.groupcollide()` function compares the rects of each element in one group with the rects of each element in another group. In this case, it compares each bullet's rect with each alien's rect and returns a dictionary containing the bullets and aliens that have collided. Each key in the

dictionary will be a bullet, and the corresponding value will be the alien that was hit. (We'll also use this dictionary when we implement a scoring system in Chapter 14.)

Add the following code to the end of `_update_bullets()` to check for collisions between bullets and aliens:

```
alien_invasion.py
def _update_bullets(self):
    """Update position of bullets and get rid of old bullets."""
    --snip--

    # Check for any bullets that have hit aliens.
    # If so, get rid of the bullet and the alien.
    collisions = pygame.sprite.groupcollide(
        self.bullets, self.aliens, True, True)
```

The new code we added compares the positions of all the bullets in `self.bullets` and all the aliens in `self.aliens`, and identifies any that overlap. Whenever the rects of a bullet and alien overlap, `groupcollide()` adds a key-value pair to the dictionary it returns. The two `True` arguments tell Pygame to delete the bullets and aliens that have collided. (To make a high-powered bullet that can travel to the top of the screen, destroying every alien in its path, you could set the first Boolean argument to `False` and keep the second Boolean argument set to `True`. The aliens hit would disappear, but all bullets would stay active until they disappeared off the top of the screen.)

When you run *Alien Invasion* now, aliens you hit should disappear. Figure 13-5 shows a fleet that has been partially shot down.



Figure 13-5: We can shoot aliens!

Making Larger Bullets for Testing

You can test many features of the game simply by running the game. But some features are tedious to test in the normal version of the game. For example, it's a lot of work to shoot down every alien on the screen multiple times to test whether your code responds to an empty fleet correctly.

To test particular features, you can change certain game settings to focus on a particular area. For example, you might shrink the screen so there are fewer aliens to shoot down or increase the bullet speed and give yourself lots of bullets at once.

My favorite change for testing *Alien Invasion* is to use really wide bullets that remain active even after they've hit an alien (see Figure 13-6). Try setting `bullet_width` to 300, or even 3000, to see how quickly you can shoot down the fleet!

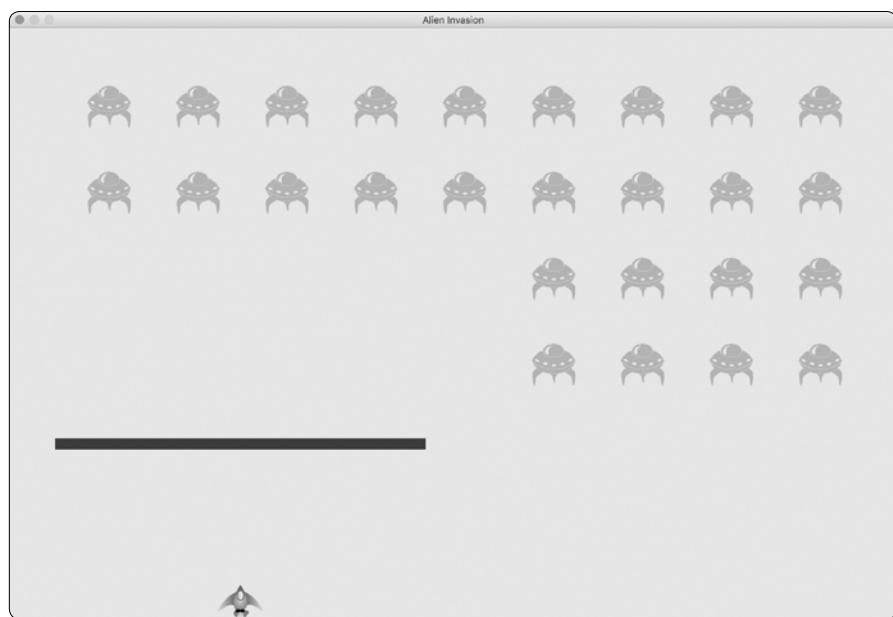


Figure 13-6: Extra-powerful bullets make some aspects of the game easier to test.

Changes like these will help you test the game more efficiently and possibly spark ideas for giving players bonus powers. Just remember to restore the settings to normal when you're finished testing a feature.

Repopulating the Fleet

One key feature of *Alien Invasion* is that the aliens are relentless: every time the fleet is destroyed, a new fleet should appear.

To make a new fleet of aliens appear after a fleet has been destroyed, we first check whether the `aliens` group is empty. If it is, we make a call to `_create_fleet()`. We'll perform this check at the end of `_update_bullets()`, because that's where individual aliens are destroyed.

```
def _update_bullets(self):
    --snip--
❶    if not self.aliens:
        # Destroy existing bullets and create new fleet.
❷    self.bullets.empty()
    self._create_fleet()
```

At ❶, we check whether the aliens group is empty. An empty group evaluates to `False`, so this is a simple way to check whether the group is empty. If it is, we get rid of any existing bullets by using the `empty()` method, which removes all the remaining sprites from a group ❷. We also call `_create_fleet()`, which fills the screen with aliens again.

Now a new fleet appears as soon as you destroy the current fleet.

Speeding Up the Bullets

If you've tried firing at the aliens in the game's current state, you might find that the bullets aren't traveling at the best speed for gameplay. They might be a little slow on your system or way too fast. At this point, you can modify the settings to make the gameplay interesting and enjoyable on your system.

We modify the speed of the bullets by adjusting the value of `bullet_speed` in `settings.py`. On my system, I'll adjust the value of `bullet_speed` to 1.5, so the bullets travel a little faster:

```
# Bullet settings
self.bullet_speed = 1.5
self.bullet_width = 3
--snip--
```

The best value for this setting depends on your system's speed, so find a value that works for you. You can adjust other settings as well.

Refactoring `_update_bullets()`

Let's refactor `_update_bullets()` so it's not doing so many different tasks. We'll move the code for dealing with bullet and alien collisions to a separate method:

```
def _update_bullets(self):
    --snip--
    # Get rid of bullets that have disappeared.
    for bullet in self.bullets.copy():
        if bullet.rect.bottom <= 0:
            self.bullets.remove(bullet)

    self._check_bullet_alien_collisions()

def _check_bullet_alien_collisions(self):
    """Respond to bullet-alien collisions."""
    # Remove any bullets and aliens that have collided.
```

```
collisions = pygame.sprite.groupcollide(  
    self.bullets, self.aliens, True, True)  
  
if not self.aliens:  
    # Destroy existing bullets and create new fleet.  
    self.bullets.empty()  
    self._create_fleet()
```

We've created a new method, `_check_bullet_alien_collisions()`, to look for collisions between bullets and aliens, and to respond appropriately if the entire fleet has been destroyed. Doing so keeps `_update_bullets()` from growing too long and simplifies further development.

TRY IT YOURSELF

13-5. Sideways Shooter Part 2: We've come a long way since Exercise 12-6, Sideways Shooter. For this exercise, try to develop Sideways Shooter to the same point we've brought *Alien Invasion* to. Add a fleet of aliens, and make them move sideways toward the ship. Or, write code that places aliens at random positions along the right side of the screen and then sends them toward the ship. Also, write code that makes the aliens disappear when they're hit.

Ending the Game

What's the fun and challenge in a game if you can't lose? If the player doesn't shoot down the fleet quickly enough, we'll have the aliens destroy the ship when they make contact. At the same time, we'll limit the number of ships a player can use, and we'll destroy the ship when an alien reaches the bottom of the screen. The game will end when the player has used up all their ships.

Detecting Alien and Ship Collisions

We'll start by checking for collisions between aliens and the ship so we can respond appropriately when an alien hits it. We'll check for alien and ship collisions immediately after updating the position of each alien in `AlienInvasion`:

```
alien_invasion.py  
def _update.aliens(self):  
    --snip--  
    self.aliens.update()  
  
    # Look for alien-ship collisions.  
    ❶    if pygame.sprite.spritecollideany(self.ship, self.aliens):  
        print("Ship hit!!!")
```

The `spritecollideany()` function takes two arguments: a sprite and a group. The function looks for any member of the group that has collided with the sprite and stops looping through the group as soon as it finds one member that has collided with the sprite. Here, it loops through the group `aliens` and returns the first alien it finds that has collided with `ship`.

If no collisions occur, `spritecollideany()` returns `None` and the `if` block at ❶ won't execute. If it finds an alien that has collided with the ship, it returns that alien and the `if` block executes: it prints *Ship hit!!!* ❷. When an alien hits the ship, we'll need to do a number of tasks: we'll need to delete all remaining aliens and bullets, recenter the ship, and create a new fleet. Before we write code to do all this, we need to know that our approach for detecting alien and ship collisions works correctly. Writing a `print()` call is a simple way to ensure we're detecting these collisions properly.

Now when you run *Alien Invasion*, the message *Ship hit!!!* should appear in the terminal whenever an alien runs into the ship. When you're testing this feature, set `alien_drop_speed` to a higher value, such as 50 or 100, so the aliens reach your ship faster.

Responding to Alien and Ship Collisions

Now we need to figure out exactly what will happen when an alien collides with the ship. Instead of destroying the `ship` instance and creating a new one, we'll count how many times the ship has been hit by tracking statistics for the game. Tracking statistics will also be useful for scoring.

Let's write a new class, `GameStats`, to track game statistics, and save it as `game_stats.py`:

```
game_stats.py
class GameStats:
    """Track statistics for Alien Invasion."""

    def __init__(self, ai_game):
        """Initialize statistics."""
        self.settings = ai_game.settings
        self.reset_stats()

    def reset_stats(self):
        """Initialize statistics that can change during the game."""
        self.ships_left = self.settings.ship_limit
```

We'll make one `GameStats` instance for the entire time *Alien Invasion* is running. But we'll need to reset some statistics each time the player starts a new game. To do this, we'll initialize most of the statistics in the `reset_stats()` method instead of directly in `__init__()`. We'll call this method from `__init__()` so the statistics are set properly when the `GameStats` instance is first created ❶. But we'll also be able to call `reset_stats()` any time the player starts a new game.

Right now we have only one statistic, `ships_left`, the value of which will change throughout the game. The number of ships the player starts with should be stored in `settings.py` as `ship_limit`:

`settings.py`

```
# Ship settings
self.ship_speed = 1.5
self.ship_limit = 3
```

We also need to make a few changes in `alien_invasion.py` to create an instance of `GameStats`. First, we'll update the `import` statements at the top of the file:

`alien_invasion.py`

```
import sys
from time import sleep

import pygame

from settings import Settings
from game_stats import GameStats
from ship import Ship
--snip--
```

We import the `sleep()` function from the `time` module in the Python standard library so we can pause the game for a moment when the ship is hit. We also import `GameStats`.

We'll create an instance of `GameStats` in `__init__()`:

`alien_invasion.py`

```
def __init__(self):
    --snip--
    self.screen = pygame.display.set_mode(
        (self.settings.screen_width, self.settings.screen_height))
    pygame.display.set_caption("Alien Invasion")

    # Create an instance to store game statistics.
    self.stats = GameStats(self)

    self.ship = Ship(self)
    --snip--
```

We make the instance after creating the game window but before defining other game elements, such as the ship.

When an alien hits the ship, we'll subtract one from the number of ships left, destroy all existing aliens and bullets, create a new fleet, and reposition the ship in the middle of the screen. We'll also pause the game for a moment so the player can notice the collision and regroup before a new fleet appears.

Let's put most of this code in a new method called `_ship_hit()`. We'll call this method from `_update.aliens()` when an alien hits the ship:

`alien_invasion.py`

```
def _ship_hit(self):
    """Respond to the ship being hit by an alien."""
```

```
❶     # Decrement ships_left.
❷     self.stats.ships_left -= 1

❸     # Get rid of any remaining aliens and bullets.
❹     self.aliens.empty()
     self.bullets.empty()

❺     # Create a new fleet and center the ship.
❻     self._create_fleet()
     self.ship.center_ship()

❾     # Pause.
❿     sleep(0.5)
```

The new method `_ship_hit()` coordinates the response when an alien hits a ship. Inside `_ship_hit()`, the number of ships left is reduced by 1 at ❶, after which we empty the groups `aliens` and `bullets` ❷.

Next, we create a new fleet and center the ship ❸. (We'll add the method `center_ship()` to `Ship` in a moment.) Then we add a pause after the updates have been made to all the game elements but before any changes have been drawn to the screen, so the player can see that their ship has been hit ❽. The `sleep()` call pauses program execution for half a second, long enough for the player to see that the alien has hit the ship. When the `sleep()` function ends, code execution moves on to the `_update_screen()` method, which draws the new fleet to the screen.

In `_update_aliens()`, we replace the `print()` call with a call to `_ship_hit()` when an alien hits the ship:

alien_invasion.py

```
def _update_aliens(self):
    --snip--
    if pygame.sprite.spritecollideany(self.ship, self.aliens):
        self._ship_hit()
```

Here's the new method `center_ship()`; add it to the end of `ship.py`:

ship.py

```
def center_ship(self):
    """Center the ship on the screen."""
    self.rect.midbottom = self.screen_rect.midbottom
    self.x = float(self.rect.x)
```

We center the ship the same way we did in `__init__()`. After centering it, we reset the `self.x` attribute, which allows us to track the ship's exact position.

NOTE

Notice that we never make more than one ship; we make only one ship instance for the whole game and recenter it whenever the ship has been hit. The statistic `ships_left` will tell us when the player has run out of ships.

Run the game, shoot a few aliens, and let an alien hit the ship. The game should pause, and a new fleet should appear with the ship centered at the bottom of the screen again.

Aliens that Reach the Bottom of the Screen

If an alien reaches the bottom of the screen, we'll have the game respond the same way it does when an alien hits the ship. To check when this happens, add a new method in *alien_invasion.py*:

alien_invasion.py

```
def _check_aliens_bottom(self):
    """Check if any aliens have reached the bottom of the screen."""
    screen_rect = self.screen.get_rect()
    for alien in self.aliens.sprites():
       ❶        if alien.rect.bottom >= screen_rect.bottom:
            # Treat this the same as if the ship got hit.
            self._ship_hit()
            break
```

The method `_check_aliens_bottom()` checks whether any aliens have reached the bottom of the screen. An alien reaches the bottom when its `rect.bottom` value is greater than or equal to the screen's `rect.bottom` attribute ❶. If an alien reaches the bottom, we call `_ship_hit()`. If one alien hits the bottom, there's no need to check the rest, so we break out of the loop after calling `_ship_hit()`.

We'll call this method from `_update_aliens()`:

alien_invasion.py

```
def _update_aliens(self):
    --snip--
    # Look for alien-ship collisions.
    if pygame.sprite.spritecollideany(self.ship, self.aliens):
        self._ship_hit()

    # Look for aliens hitting the bottom of the screen.
    self._check_aliens_bottom()
```

We call `_check_aliens_bottom()` after updating the positions of all the aliens and after looking for alien and ship collisions ❷. Now a new fleet will appear every time the ship is hit by an alien or an alien reaches the bottom of the screen.

Game Over!

Alien Invasion feels more complete now, but the game never ends. The value of `ships_left` just grows increasingly negative. Let's add a `game_active` flag as an attribute to `GameStats` to end the game when the player runs out of ships. We'll set this flag at the end of the `__init__()` method in `GameStats`:

game_stats.py

```
def __init__(self, ai_game):
    --snip--
    # Start Alien Invasion in an active state.
    self.game_active = True
```

Now we add code to `_ship_hit()` that sets `game_active` to `False` when the player has used up all their ships:

alien_invasion.py

```
def _ship_hit(self):
    """Respond to ship being hit by alien."""
    if self.stats.ships_left > 0:
        # Decrement ships_left.
        self.stats.ships_left -= 1
        --snip--
        # Pause.
        sleep(0.5)
    else:
        self.stats.game_active = False
```

Most of `_ship_hit()` is unchanged. We've moved all the existing code into an `if` block, which tests to make sure the player has at least one ship remaining. If so, we create a new fleet, pause, and move on. If the player has no ships left, we set `game_active` to `False`.

Identifying When Parts of the Game Should Run

We need to identify the parts of the game that should always run and the parts that should run only when the game is active:

alien_invasion.py

```
def run_game(self):
    """Start the main loop for the game."""
    while True:
        self._check_events()

        if self.stats.game_active:
            self.ship.update()
            self._update_bullets()
            self._update.aliens()

        self._update_screen()
```

In the main loop, we always need to call `_check_events()`, even if the game is inactive. For example, we still need to know if the user presses Q to quit the game or clicks the button to close the window. We also continue updating the screen so we can make changes to the screen while waiting to see whether the player chooses to start a new game. The rest of the function calls only need to happen when the game is active, because when the game is inactive, we don't need to update the positions of game elements.

Now when you play *Alien Invasion*, the game should freeze when you've used up all your ships.

TRY IT YOURSELF

13-6. Game Over: In Sideways Shooter, keep track of the number of times the ship is hit and the number of times an alien is hit by the ship. Decide on an appropriate condition for ending the game, and stop the game when this situation occurs.

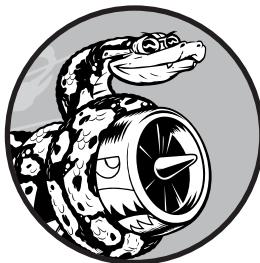
Summary

In this chapter, you learned how to add a large number of identical elements to a game by creating a fleet of aliens. You used nested loops to create a grid of elements, and you made a large set of game elements move by calling each element’s `update()` method. You learned to control the direction of objects on the screen and to respond to specific situations, such as when the fleet reaches the edge of the screen. You detected and responded to collisions when bullets hit aliens and aliens hit the ship. You also learned how to track statistics in a game and use a `game_active` flag to determine when the game is over.

In the next and final chapter of this project, we’ll add a Play button so the player can choose when to start their first game and whether to play again when the game ends. We’ll speed up the game each time the player shoots down the entire fleet, and we’ll add a scoring system. The final result will be a fully playable game!

14

SCORING



In this chapter, we'll finish the *Alien Invasion* game. We'll add a Play button to start a game on demand or to restart a game once it ends. We'll also change the game so it speeds up when the player moves up a level, and implement a scoring system. By the end of the chapter, you'll know enough to start writing games that increase in difficulty as a player progresses and show scores.

Adding the Play Button

In this section, we'll add a Play button that appears before a game begins and reappears when the game ends so the player can play again.

Right now the game begins as soon as you run *alien_invasion.py*. Let's start the game in an inactive state and then prompt the player to click a Play button to begin. To do this, modify the `__init__()` method of `GameStats`:

game_stats.py

```
def __init__(self, ai_game):
    """Initialize statistics."""
    self.settings = ai_game.settings
    self.reset_stats()

    # Start game in an inactive state.
    self.game_active = False
```

Now the game should start in an inactive state with no way for the player to start it until we make a Play button.

Creating a Button Class

Because Pygame doesn't have a built-in method for making buttons, we'll write a `Button` class to create a filled rectangle with a label. You can use this code to make any button in a game. Here's the first part of the `Button` class; save it as `button.py`:

button.py

```
import pygame.font

class Button:

    ❶    def __init__(self, ai_game, msg):
        """Initialize button attributes."""
        self.screen = ai_game.screen
        self.screen_rect = self.screen.get_rect()

        # Set the dimensions and properties of the button.
    ❷        self.width, self.height = 200, 50
        self.button_color = (0, 255, 0)
        self.text_color = (255, 255, 255)
    ❸        self.font = pygame.font.SysFont(None, 48)

        # Build the button's rect object and center it.
    ❹        self.rect = pygame.Rect(0, 0, self.width, self.height)
        self.rect.center = self.screen_rect.center

        # The button message needs to be prepped only once.
    ❺        self._prep_msg(msg)
```

First, we import the `pygame.font` module, which lets Pygame render text to the screen. The `__init__()` method takes the parameters `self`, the `ai_game`

object, and `msg`, which contains the button's text ❶. We set the button dimensions at ❷, and then set `button_color` to color the button's `rect` object bright green and set `text_color` to render the text in white.

At ❸, we prepare a font attribute for rendering text. The `None` argument tells Pygame to use the default font, and `48` specifies the size of the text. To center the button on the screen, we create a `rect` for the button ❹ and set its `center` attribute to match that of the screen.

Pygame works with text by rendering the string you want to display as an image. At ❺, we call `_prep_msg()` to handle this rendering.

Here's the code for `_prep_msg()`:

```
button.py    def _prep_msg(self, msg):
    """Turn msg into a rendered image and center text on the button."""
❶    self.msg_image = self.font.render(msg, True, self.text_color,
                                         self.button_color)
❷    self.msg_image_rect = self.msg_image.get_rect()
    self.msg_image_rect.center = self.rect.center
```

The `_prep_msg()` method needs a `self` parameter and the text to be rendered as an image (`msg`). The call to `font.render()` turns the text stored in `msg` into an image, which we then store in `self.msg_image` ❶. The `font.render()` method also takes a Boolean value to turn antialiasing on or off (antialiasing makes the edges of the text smoother). The remaining arguments are the specified font color and background color. We set antialiasing to `True` and set the text background to the same color as the button. (If you don't include a background color, Pygame will try to render the font with a transparent background.)

At ❷, we center the text image on the button by creating a `rect` from the image and setting its `center` attribute to match that of the button.

Finally, we create a `draw_button()` method that we can call to display the button onscreen:

```
button.py    def draw_button(self):
    # Draw blank button and then draw message.
    self.screen.fill(self.button_color, self.rect)
    self.screen.blit(self.msg_image, self.msg_image_rect)
```

We call `screen.fill()` to draw the rectangular portion of the button. Then we call `screen.blit()` to draw the text image to the screen, passing it an image and the `rect` object associated with the image. This completes the `Button` class.

Drawing the Button to the Screen

We'll use the `Button` class to create a Play button in `AlienInvasion`. First, we'll update the `import` statements:

```
alien_invasion.py    --snip--
from game_stats import GameStats
from button import Button
```

Because we need only one Play button, we'll create the button in the `_init_()` method of `AlienInvasion`. We can place this code at the very end of `_init_()`:

alien_invasion.py

```
def __init__(self):
    --snip--
    self._create_fleet()

    # Make the Play button.
    self.play_button = Button(self, "Play")
```

This code creates an instance of `Button` with the label *Play*, but it doesn't draw the button to the screen. We'll call the button's `draw_button()` method in `_update_screen()`:

alien_invasion.py

```
def _update_screen(self):
    --snip--
    self.aliens.draw(self.screen)

    # Draw the play button if the game is inactive.
    if not self.stats.game_active:
        self.play_button.draw_button()

pygame.display.flip()
```

To make the Play button visible above all other elements on the screen, we draw it after all the other elements have been drawn but before flipping to a new screen. We include it in an `if` block, so the button only appears when the game is inactive.

Now when you run *Alien Invasion*, you should see a Play button in the center of the screen, as shown in Figure 14-1.

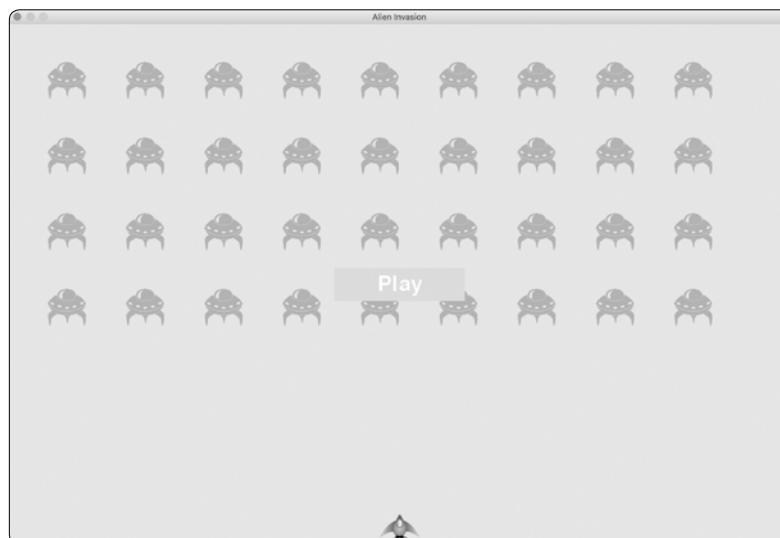


Figure 14-1: A Play button appears when the game is inactive.

Starting the Game

To start a new game when the player clicks Play, add the following `elif` block to the end of `_check_events()` to monitor mouse events over the button:

alien_invasion.py

```
def _check_events(self):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            --snip--
①    elif event.type == pygame.MOUSEBUTTONDOWN:
②        mouse_pos = pygame.mouse.get_pos()
③        self._check_play_button(mouse_pos)
```

Pygame detects a `MOUSEBUTTONDOWN` event when the player clicks anywhere on the screen ①, but we want to restrict our game to respond to mouse clicks only on the Play button. To accomplish this, we use `pygame.mouse.get_pos()`, which returns a tuple containing the mouse cursor's x- and y-coordinates when the mouse button is clicked ②. We send these values to the new method `_check_play_button()` ③.

Here's `_check_play_button()`, which I chose to place after `_check_events()`:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    """Start a new game when the player clicks Play."""
①    if self.play_button.rect.collidepoint(mouse_pos):
        self.stats.game_active = True
```

We use the `rect` method `collidepoint()` to check whether the point of the mouse click overlaps the region defined by the Play button's `rect` ①. If so, we set `game_active` to `True`, and the game begins!

At this point, you should be able to start and play a full game. When the game ends, the value of `game_active` should become `False` and the Play button should reappear.

Resetting the Game

The Play button code we just wrote works the first time the player clicks Play. But it doesn't work after the first game ends, because the conditions that caused the game to end haven't been reset.

To reset the game each time the player clicks Play, we need to reset the game statistics, clear out the old aliens and bullets, build a new fleet, and center the ship, as shown here:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    """Start a new game when the player clicks Play."""
    if self.play_button.rect.collidepoint(mouse_pos):
        # Reset the game statistics.
①        self.stats.reset_stats()
        self.stats.game_active = True

②        # Get rid of any remaining aliens and bullets.
        self.aliens.empty()
```

```
    self.bullets.empty()

    # Create a new fleet and center the ship.
❸    self._create_fleet()
        self.ship.center_ship()
```

At ❶, we reset the game statistics, which gives the player three new ships. Then we set `game_active` to `True` so the game will begin as soon as the code in this function finishes running. We empty the `aliens` and `bullets` groups ❷, and then create a new fleet and center the ship ❸.

Now the game will reset properly each time you click Play, allowing you to play it as many times as you want!

Deactivating the Play Button

One issue with our Play button is that the button region on the screen will continue to respond to clicks even when the Play button isn't visible. If you click the Play button area by accident after a game begins, the game will restart!

To fix this, set the game to start only when `game_active` is `False`:

```
alien_invasion.py
def _check_play_button(self, mouse_pos):
    """Start a new game when the player clicks Play."""
❶    button_clicked = self.play_button.rect.collidepoint(mouse_pos)
❷    if button_clicked and not self.stats.game_active:
        # Reset the game statistics.
        self.stats.reset_stats()
        --snip--
```

The flag `button_clicked` stores a `True` or `False` value ❶, and the game will restart only if Play is clicked *and* the game is not currently active ❷. To test this behavior, start a new game and repeatedly click where the Play button should be. If everything works as expected, clicking the Play button area should have no effect on the gameplay.

Hiding the Mouse Cursor

We want the mouse cursor to be visible to begin play, but once play begins, it just gets in the way. To fix this, we'll make it invisible when the game becomes active. We can do this at the end of the `if` block in `_check_play_button()`:

```
alien_invasion.py
def _check_play_button(self, mouse_pos):
    """Start a new game when the player clicks Play."""
    button_clicked = self.play_button.rect.collidepoint(mouse_pos)
    if button_clicked and not self.stats.game_active:
        --snip--
        # Hide the mouse cursor.
        pygame.mouse.set_visible(False)
```

Passing `False` to `set_visible()` tells Pygame to hide the cursor when the mouse is over the game window.

We'll make the cursor reappear once the game ends so the player can click Play again to begin a new game. Here's the code to do that:

alien_invasion.py

```
def _ship_hit(self):
    """Respond to ship being hit by alien."""
    if self.stats.ships_left > 0:
        --snip--
    else:
        self.stats.game_active = False
        pygame.mouse.set_visible(True)
```

We make the cursor visible again as soon as the game becomes inactive, which happens in `_ship_hit()`. Attention to details like this makes your game more professional looking and allows the player to focus on playing rather than figuring out the user interface.

TRY IT YOURSELF

14-1. Press P to Play: Because *Alien Invasion* uses keyboard input to control the ship, it would be useful to start the game with a keypress. Add code that lets the player press P to start. It might help to move some code from `_check_play_button()` to a `_start_game()` method that can be called from `_check_play_button()` and `_check_keydown_events()`.

14-2. Target Practice: Create a rectangle at the right edge of the screen that moves up and down at a steady rate. Then have a ship appear on the left side of the screen that the player can move up and down while firing bullets at the moving, rectangular target. Add a Play button that starts the game, and when the player misses the target three times, end the game and make the Play button reappear. Let the player restart the game with this Play button.

Leveling Up

In our current game, once a player shoots down the entire alien fleet, the player reaches a new level, but the game difficulty doesn't change. Let's liven things up a bit and make the game more challenging by increasing the game's speed each time a player clears the screen.

Modifying the Speed Settings

We'll first reorganize the `Settings` class to group the game settings into static and changing ones. We'll also make sure that settings that change

during the game reset when we start a new game. Here's the `_init_()` method for `settings.py`:

```
settings.py      def __init__(self):
                  """Initialize the game's static settings."""
                  # Screen settings
                  self.screen_width = 1200
                  self.screen_height = 800
                  self.bg_color = (230, 230, 230)

                  # Ship settings
                  self.ship_limit = 3

                  # Bullet settings
                  self.bullet_width = 3
                  self.bullet_height = 15
                  self.bullet_color = 60, 60, 60
                  self.bullets_allowed = 3

                  # Alien settings
                  self.fleet_drop_speed = 10

                  # How quickly the game speeds up
❶                 self.speedup_scale = 1.1

❷                 self.initialize_dynamic_settings()
```

We continue to initialize those settings that stay constant in the `_init_()` method. At ❶, we add a `speedup_scale` setting to control how quickly the game speeds up: a value of 2 will double the game speed every time the player reaches a new level; a value of 1 will keep the speed constant. A value like 1.1 should increase the speed enough to make the game challenging but not impossible. Finally, we call the `initialize_dynamic_settings()` method to initialize the values for attributes that need to change throughout the game ❷.

Here's the code for `initialize_dynamic_settings()`:

```
settings.py      def initialize_dynamic_settings(self):
                  """Initialize settings that change throughout the game."""
                  self.ship_speed = 1.5
                  self.bullet_speed = 3.0
                  self.alien_speed = 1.0

                  # fleet_direction of 1 represents right; -1 represents left.
                  self.fleet_direction = 1
```

This method sets the initial values for the ship, bullet, and alien speeds. We'll increase these speeds as the player progresses in the game and reset them each time the player starts a new game. We include `fleet_direction` in this method so the aliens always move right at the beginning of a new game. We don't need to increase the value of `fleet_drop_speed`, because when the aliens move faster across the screen, they'll also come down the screen faster.

To increase the speeds of the ship, bullets, and aliens each time the player reaches a new level, we'll write a new method called `increase_speed()`:

settings.py

```
def increase_speed(self):
    """Increase speed settings."""
    self.ship_speed *= self.speedup_scale
    self.bullet_speed *= self.speedup_scale
    self.alien_speed *= self.speedup_scale
```

To increase the speed of these game elements, we multiply each speed setting by the value of `speedup_scale`.

We increase the game's tempo by calling `increase_speed()` in `_check_bullet_alien_collisions()` when the last alien in a fleet has been shot down:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    --snip--
    if not self.aliens:
        # Destroy existing bullets and create new fleet.
        self.bullets.empty()
        self._create_fleet()
        self.settings.increase_speed()
```

Changing the values of the speed settings `ship_speed`, `alien_speed`, and `bullet_speed` is enough to speed up the entire game!

Resetting the Speed

Now we need to return any changed settings to their initial values each time the player starts a new game; otherwise, each new game would start with the increased speed settings of the previous game:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    """Start a new game when the player clicks Play."""
    button_clicked = self.play_button.rect.collidepoint(mouse_pos)
    if button_clicked and not self.stats.game_active:
        # Reset the game settings.
        self.settings.initialize_dynamic_settings()
    --snip--
```

Playing *Alien Invasion* should be more fun and challenging now. Each time you clear the screen, the game should speed up and become slightly more difficult. If the game becomes too difficult too quickly, decrease the value of `settings.speedup_scale`. Or if the game isn't challenging enough, increase the value slightly. Find a sweet spot by ramping up the difficulty in a reasonable amount of time. The first couple of screens should be easy, the next few challenging but doable, and subsequent screens almost impossibly difficult.

TRY IT YOURSELF

14-3. Challenging Target Practice: Start with your work from Exercise 14-2 (page 285). Make the target move faster as the game progresses, and restart the target at the original speed when the player clicks Play.

14-4. Difficulty Levels: Make a set of buttons for *Alien Invasion* that allows the player to select an appropriate starting difficulty level for the game. Each button should assign the appropriate values for the attributes in `Settings` needed to create different difficulty levels.

Scoring

Let's implement a scoring system to track the game's score in real time and display the high score, level, and number of ships remaining.

The score is a game statistic, so we'll add a `score` attribute to `GameStats`:

```
game_stats.py
class GameStats:
    --snip--
    def reset_stats(self):
        """Initialize statistics that can change during the game."""
        self.ships_left = self.ai_settings.ship_limit
        self.score = 0
```

To reset the score each time a new game starts, we initialize `score` in `reset_stats()` rather than `__init__()`.

Displaying the Score

To display the score on the screen, we first create a new class, `Scoreboard`. For now, this class will just display the current score, but eventually we'll use it to report the high score, level, and number of ships remaining as well. Here's the first part of the class; save it as `scoreboard.py`:

```
scoreboard.py
import pygame.font

class Scoreboard:
    """A class to report scoring information."""

    ❶    def __init__(self, ai_game):
        """Initialize scorekeeping attributes."""
        self.screen = ai_game.screen
        self.screen_rect = self.screen.get_rect()
        self.settings = ai_game.settings
        self.stats = ai_game.stats

        # Font settings for scoring information.
        ❷    self.text_color = (30, 30, 30)
        ❸    self.font = pygame.font.SysFont(None, 48)
```

```
❸     # Prepare the initial score image.  
❹     self.prep_score()
```

Because Scoreboard writes text to the screen, we begin by importing the `pygame.font` module. Next, we give `__init__()` the `ai_game` parameter so it can access the `settings`, `screen`, and `stats` objects, which it will need to report the values we're tracking ❶. Then we set a text color ❷ and instantiate a font object ❸.

To turn the text to be displayed into an image, we call `prep_score()` ❹, which we define here:

```
scoreboard.py      def prep_score(self):  
                      """Turn the score into a rendered image."""  
❶          score_str = str(self.stats.score)  
❷          self.score_image = self.font.render(score_str, True,  
                                             self.text_color, self.settings.bg_color)  
  
          # Display the score at the top right of the screen.  
❸          self.score_rect = self.score_image.get_rect()  
❹          self.score_rect.right = self.screen_rect.right - 20  
❺          self.score_rect.top = 20
```

In `prep_score()`, we turn the numerical value `stats.score` into a string ❶, and then pass this string to `render()`, which creates the image ❷. To display the score clearly onscreen, we pass the screen's background color and the text color to `render()`.

We'll position the score in the upper-right corner of the screen and have it expand to the left as the score increases and the width of the number grows. To make sure the score always lines up with the right side of the screen, we create a rect called `score_rect` ❸ and set its right edge 20 pixels from the right edge of the screen ❹. We then place the top edge 20 pixels down from the top of the screen ❺.

Then we create a `show_score()` method to display the rendered score image:

```
scoreboard.py      def show_score(self):  
                      """Draw score to the screen."""  
                      self.screen.blit(self.score_image, self.score_rect)
```

This method draws the score image onscreen at the location `score_rect` specifies.

Making a Scoreboard

To display the score, we'll create a `Scoreboard` instance in `AlienInvasion`. First, let's update the `import` statements:

```
alien_invasion.py--snip--  
from game_stats import GameStats  
from scoreboard import Scoreboard  
--snip--
```

Next, we make an instance of Scoreboard in `__init__()`:

```
alien_invasion.py
def __init__(self):
    --snip--
    pygame.display.set_caption("Alien Invasion")

    # Create an instance to store game statistics,
    # and create a scoreboard.
    self.stats = GameStats(self)
    self.sb = Scoreboard(self)
    --snip--
```

Then we draw the scoreboard onscreen in `_update_screen()`:

```
alien_invasion.py
def _update_screen(self):
    --snip--
    self.aliens.draw(self.screen)

    # Draw the score information.
    self.sb.show_score()

    # Draw the play button if the game is inactive.
    --snip--
```

We call `show_score()` just before we draw the Play button.

When you run *Alien Invasion* now, a 0 should appear at the top right of the screen. (At this point, we just want to make sure the score appears in the right place before developing the scoring system further.) Figure 14-2 shows the score as it appears before the game starts.

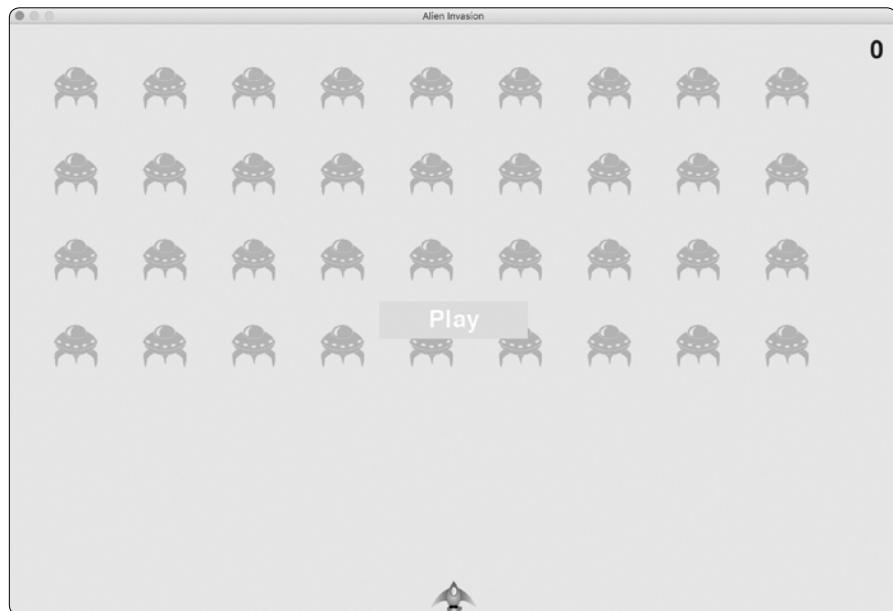


Figure 14-2: The score appears at the top-right corner of the screen.

Next, we'll assign point values to each alien!

Updating the Score as Aliens Are Shot Down

To write a live score onscreen, we update the value of `stats.score` whenever an alien is hit, and then call `prep_score()` to update the score image. But first, let's determine how many points a player gets each time they shoot down an alien:

settings.py

```
def initialize_dynamic_settings(self):
    --snip--

    # Scoring
    self.alien_points = 50
```

We'll increase each alien's point value as the game progresses. To make sure this point value is reset each time a new game starts, we set the value in `initialize_dynamic_settings()`.

Let's update the score each time an alien is shot down in `_check_bullet_alien_collisions()`:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    """Respond to bullet-alien collisions."""
    # Remove any bullets and aliens that have collided.
    collisions = pygame.sprite.groupcollide(
        self.bullets, self.aliens, True, True)

    if collisions:
        self.stats.score += self.settings.alien_points
        self.sb.prep_score()
    --snip--
```

When a bullet hits an alien, Pygame returns a `collisions` dictionary. We check whether the dictionary exists, and if it does, the alien's value is added to the score. We then call `prep_score()` to create a new image for the updated score.

Now when you play *Alien Invasion*, you should be able to rack up points!

Resetting the Score

Right now, we're only prepping a new score *after* an alien has been hit, which works for most of the game. But we still see the old score when a new game starts until the first alien is hit in the new game.

We can fix this by prepping the score when starting a new game:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    --snip--
    if button_clicked and not self.stats.game_active:
        --snip--
        # Reset the game statistics.
        self.stats.reset_stats()
```

```
self.stats.game_active = True  
self.sb.prep_score()  
--snip--
```

We call `prep_score()` after resetting the game stats when starting a new game. This preps the scoreboard with a 0 score.

Making Sure to Score All Hits

As currently written, our code could miss scoring for some aliens. For example, if two bullets collide with aliens during the same pass through the loop or if we make an extra-wide bullet to hit multiple aliens, the player will only receive points for hitting one of the aliens. To fix this, let's refine the way that bullet and alien collisions are detected.

In `_check_bullet_alien_collisions()`, any bullet that collides with an alien becomes a key in the `collisions` dictionary. The value associated with each bullet is a list of aliens it has collided with. We loop through the values in the `collisions` dictionary to make sure we award points for each alien hit:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):  
    --snip--  
    if collisions:  
        ❶     for aliens in collisions.values():  
            self.stats.score += self.settings.alien_points * len(aliens)  
            self.sb.prep_score()  
    --snip--
```

If the `collisions` dictionary has been defined, we loop through all values in the dictionary. Remember that each value is a list of aliens hit by a single bullet. We multiply the value of each alien by the number of aliens in each list and add this amount to the current score. To test this, change the width of a bullet to 300 pixels and verify that you receive points for each alien you hit with your extra-wide bullets; then return the bullet width to its normal value.

Increasing Point Values

Because the game gets more difficult each time a player reaches a new level, aliens in later levels should be worth more points. To implement this functionality, we'll add code to increase the point value when the game's speed increases:

settings.py

```
class Settings:  
    """A class to store all settings for Alien Invasion."""  
  
    def __init__(self):  
        --snip--  
        # How quickly the game speeds up  
        self.speedup_scale = 1.1
```

```
# How quickly the alien point values increase
❶ self.score_scale = 1.5

self.initialize_dynamic_settings()

def initialize_dynamic_settings(self):
    --snip--

def increase_speed(self):
    """Increase speed settings and alien point values."""
    self.ship_speed *= self.speedup_scale
    self.bullet_speed *= self.speedup_scale
    self.alien_speed *= self.speedup_scale

❷ self.alien_points = int(self.alien_points * self.score_scale)
```

We define a rate at which points increase, which we call `score_scale` ❶. A small increase in speed (1.1) makes the game more challenging quickly. But to see a more notable difference in scoring, we need to change the alien point value by a larger amount (1.5). Now when we increase the game's speed, we also increase the point value of each hit ❷. We use the `int()` function to increase the point value by whole integers.

To see the value of each alien, add a `print()` call to the `increase_speed()` method in `Settings`:

```
settings.py
def increase_speed(self):
    --snip--
    self.alien_points = int(self.alien_points * self.score_scale)
    print(self.alien_points)
```

The new point value should appear in the terminal every time you reach a new level.

NOTE

Be sure to remove the `print()` call after verifying that the point value is increasing, or it might affect your game's performance and distract the player.

Rounding the Score

Most arcade-style shooting games report scores as multiples of 10, so let's follow that lead with our scores. Also, let's format the score to include comma separators in large numbers. We'll make this change in `Scoreboard`:

```
scoreboard.py
def prep_score(self):
    """Turn the score into a rendered image."""
❶    rounded_score = round(self.stats.score, -1)
❷    score_str = "{:,}".format(rounded_score)
    self.score_image = self.font.render(score_str, True,
                                         self.text_color, self.settings.bg_color)
    --snip--
```

The `round()` function normally rounds a decimal number to a set number of decimal places given as the second argument. However, when you pass a negative number as the second argument, `round()` will round the value to the nearest 10, 100, 1000, and so on. The code at ❶ tells Python to round the value of `stats.score` to the nearest 10 and store it in `rounded_score`.

At ❷, a string formatting directive tells Python to insert commas into numbers when converting a numerical value to a string: for example, to output 1,000,000 instead of 1000000. Now when you run the game, you should see a neatly formatted, rounded score even when you rack up lots of points, as shown in Figure 14-3.

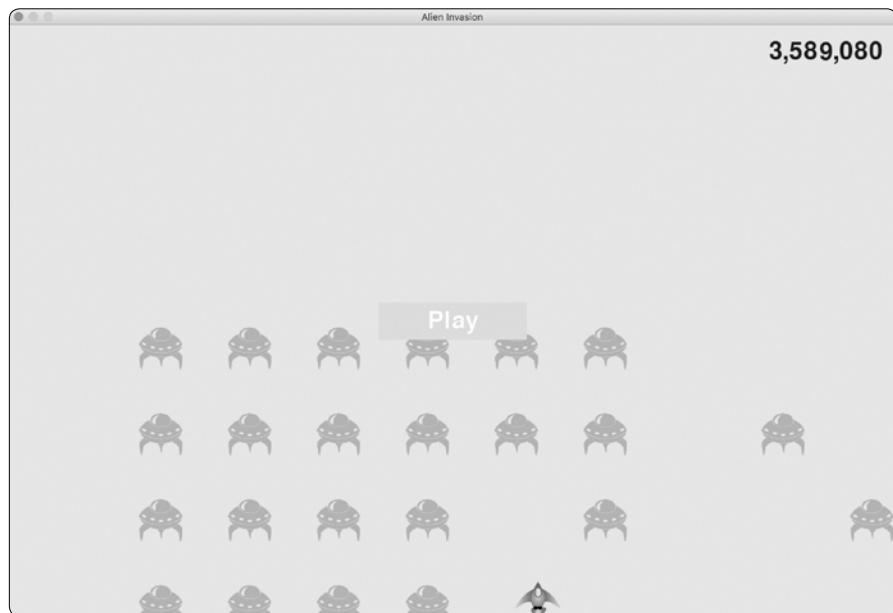


Figure 14-3: A rounded score with comma separators

High Scores

Every player wants to beat a game's high score, so let's track and report high scores to give players something to work toward. We'll store high scores in `GameStats`:

```
game_stats.py      def __init__(self, ai_game):
                  --snip--
                  # High score should never be reset.
                  self.high_score = 0
```

Because the high score should never be reset, we initialize `high_score` in `__init__()` rather than in `reset_stats()`.

Next, we'll modify `Scoreboard` to display the high score. Let's start with the `__init__()` method:

```
scoreboard.py    def __init__(self, ai_game):
                  --snip--
                  # Prepare the initial score images.
                  self.prep_score()
❶      self.prep_high_score()
```

The high score will be displayed separately from the score, so we need a new method, `prep_high_score()`, to prepare the high score image ❶.

Here's the `prep_high_score()` method:

```
scoreboard.py    def prep_high_score(self):
                  """Turn the high score into a rendered image."""
❶      high_score = round(self.stats.high_score, -1)
                  high_score_str = "{:,}".format(high_score)
❷      self.high_score_image = self.font.render(high_score_str, True,
                                                self.text_color, self.settings.bg_color)

                  # Center the high score at the top of the screen.
                  self.high_score_rect = self.high_score_image.get_rect()
❸      self.high_score_rect.centerx = self.screen_rect.centerx
❹      self.high_score_rect.top = self.score_rect.top
```

We round the high score to the nearest 10 and format it with commas ❶. We then generate an image from the high score ❷, center the high score rect horizontally ❸, and set its `top` attribute to match the top of the score image ❹.

The `show_score()` method now draws the current score at the top right and the high score at the top center of the screen:

```
scoreboard.py    def show_score(self):
                  """Draw score to the screen."""
                  self.screen.blit(self.score_image, self.score_rect)
                  self.screen.blit(self.high_score_image, self.high_score_rect)
```

To check for high scores, we'll write a new method, `check_high_score()`, in `Scoreboard`:

```
scoreboard.py    def check_high_score(self):
                  """Check to see if there's a new high score."""
                  if self.stats.score > self.stats.high_score:
                      self.stats.high_score = self.stats.score
                      self.prep_high_score()
```

The method `check_high_score()` checks the current score against the high score. If the current score is greater, we update the value of `high_score` and call `prep_high_score()` to update the high score's image.

We need to call `check_high_score()` each time an alien is hit after updating the score in `_check_bullet_alien_collisions()`:

`alien_invasion.py`

```
def _check_bullet_alien_collisions(self):
    --snip--
    if collisions:
        for aliens in collisions.values():
            self.stats.score += self.settings.alien_points * len(aliens)
            self.sb.prep_score()
            self.sb.check_high_score()
    --snip--
```

We call `check_high_score()` when the `collisions` dictionary is present, and we do so after updating the score for all the aliens that have been hit.

The first time you play *Alien Invasion*, your score will be the high score, so it will be displayed as the current score and the high score. But when you start a second game, your high score should appear in the middle and your current score at the right, as shown in Figure 14-4.

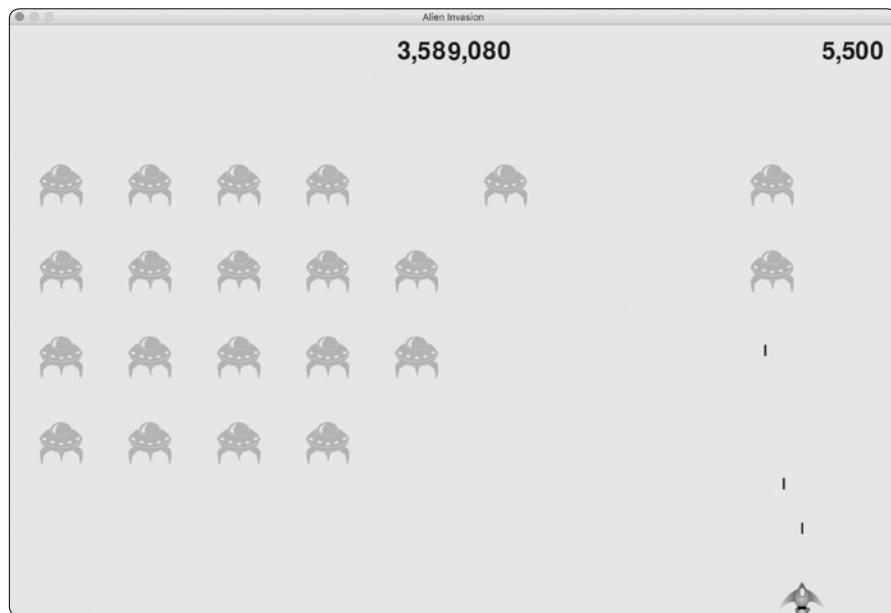


Figure 14-4: The high score is shown at the top center of the screen.

Displaying the Level

To display the player's level in the game, we first need an attribute in `GameStats` representing the current level. To reset the level at the start of each new game, initialize it in `reset_stats()`:

`game_stats.py`

```
def reset_stats(self):
    """Initialize statistics that can change during the game."""
    self.ships_left = self.settings.ship_limit
```

```
    self.score = 0
    self.level = 1
```

To have Scoreboard display the current level, we call a new method, `prep_level()`, from `__init__()`:

```
scoreboard.py
def __init__(self, ai_game):
    --snip--
    self.prep_high_score()
    self.prep_level()
```

Here's `prep_level()`:

```
scoreboard.py
def prep_level(self):
    """Turn the level into a rendered image."""
    level_str = str(self.stats.level)
❶    self.level_image = self.font.render(level_str, True,
                                           self.text_color, self.settings.bg_color)

    # Position the level below the score.
    self.level_rect = self.level_image.get_rect()
❷    self.level_rect.right = self.score_rect.right
❸    self.level_rect.top = self.score_rect.bottom + 10
```

The `prep_level()` method creates an image from the value stored in `stats.level` ❶ and sets the image's `right` attribute to match the score's `right` attribute ❷. It then sets the `top` attribute 10 pixels beneath the bottom of the score image to leave space between the score and the level ❸.

We also need to update `show_score()`:

```
scoreboard.py
def show_score(self):
    """Draw scores and level to the screen."""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
    self.screen.blit(self.level_image, self.level_rect)
```

This new line draws the level image to the screen.

We'll increment `stats.level` and update the level image in `_check_bullet_alien_collisions()`:

```
alien_invasion.py
def _check_bullet_alien_collisions(self):
    --snip--
    if not self.aliens:
        # Destroy existing bullets and create new fleet.
        self.bullets.empty()
        self._create_fleet()
        self.settings.increase_speed()

        # Increase level.
        self.stats.level += 1
        self.sb.prep_level()
```

If a fleet is destroyed, we increment the value of `stats.level` and call `prep_level()` to make sure the new level displays correctly.

To ensure the level image updates properly at the start of a new game, we also call `prep_level()` when the player clicks the Play button:

`alien_invasion.py`

```
def _check_play_button(self, mouse_pos):
    --snip--
    if button_clicked and not self.stats.game_active:
        --snip--
        self.sb.prep_score()
        self.sb.prep_level()
        --snip--
```

We call `prep_level()` right after calling `prep_score()`.

Now you'll see how many levels you've completed, as shown in Figure 14-5.

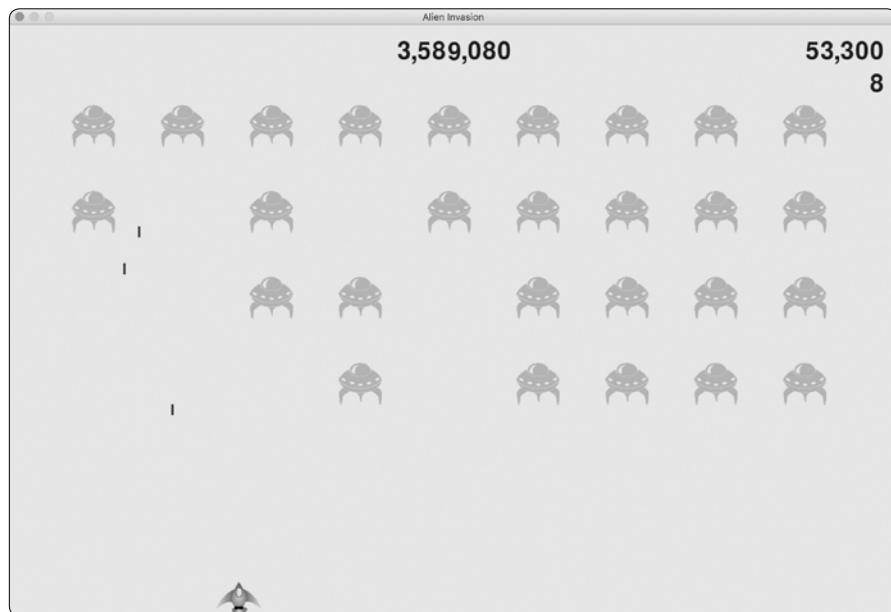


Figure 14-5: The current level appears just below the current score.

NOTE

In some classic games, the scores have labels, such as *Score*, *High Score*, and *Level*. We've omitted these labels because the meaning of each number becomes clear once you've played the game. To include these labels, add them to the score strings just before the calls to `font.render()` in `Scoreboard`.

Displaying the Number of Ships

Finally, let's display the number of ships the player has left, but this time, let's use a graphic. To do so, we'll draw ships in the upper-left corner of

the screen to represent how many ships are left, just as many classic arcade games do.

First, we need to make `Ship` inherit from `Sprite` so we can create a group of ships:

`ship.py`

```
import pygame
from pygame.sprite import Sprite

❶ class Ship(Sprite):
    """A class to manage the ship."""

    def __init__(self, ai_game):
        """Initialize the ship and set its starting position."""
❷     super().__init__()
        --snip--
```

Here we import `Sprite`, make sure `Ship` inherits from `Sprite` ❶, and call `super()` at the beginning of `__init__()` ❷.

Next, we need to modify `Scoreboard` to create a group of ships we can display. Here are the `import` statements for `Scoreboard`:

`scoreboard.py`

```
import pygame.font
from pygame.sprite import Group

from ship import Ship
```

Because we're making a group of ships, we import the `Group` and `Ship` classes.

Here's `__init__()`:

`scoreboard.py`

```
def __init__(self, ai_game):
    """Initialize scorekeeping attributes."""
    self.ai_game = ai_game
    self.screen = ai_game.screen
    --snip--
    self.prep_level()
    self.prep_ships()
```

We assign the game instance to an attribute, because we'll need it to create some ships. We call `prep_ships()` after the call to `prep_level()`.

Here's `prep_ships()`:

`scoreboard.py`

```
def prep_ships(self):
    """Show how many ships are left."""
❶    self.ships = Group()
❷    for ship_number in range(self.stats.ships_left):
        ship = Ship(self.ai_game)
❸        ship.rect.x = 10 + ship_number * ship.rect.width
❹        ship.rect.y = 10
❺        self.ships.add(ship)
```

The `prep_ships()` method creates an empty group, `self.ships`, to hold the ship instances ❶. To fill this group, a loop runs once for every ship the player has left ❷. Inside the loop, we create a new ship and set each ship's x-coordinate value so the ships appear next to each other with a 10-pixel margin on the left side of the group of ships ❸. We set the y-coordinate value 10 pixels down from the top of the screen so the ships appear in the upper-left corner of the screen ❹. Then we add each new ship to the group `ships` ❺.

Now we need to draw the ships to the screen:

scoreboard.py

```
def show_score(self):
    """Draw scores, level, and ships to the screen."""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
    self.screen.blit(self.level_image, self.level_rect)
    self.ships.draw(self.screen)
```

To display the ships on the screen, we call `draw()` on the group, and Pygame draws each ship.

To show the player how many ships they have to start with, we call `prep_ships()` when a new game starts. We do this in `_check_play_button()` in `AlienInvasion`:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    --snip--
    if button_clicked and not self.stats.game_active:
        --snip--
        self.sb.prep_score()
        self.sb.prep_level()
        self.sb.prep_ships()
    --snip--
```

We also call `prep_ships()` when a ship is hit to update the display of ship images when the player loses a ship:

alien_invasion.py

```
def _ship_hit(self):
    """Respond to ship being hit by alien."""
    if self.stats.ships_left > 0:
        # Decrement ships_left, and update scoreboard.
        self.stats.ships_left -= 1
        self.sb.prep_ships()
    --snip--
```

We call `prep_ships()` after decreasing the value of `ships_left`, so the correct number of ships displays each time a ship is destroyed.

Figure 14-6 shows the complete scoring system with the remaining ships displayed at the top left of the screen.

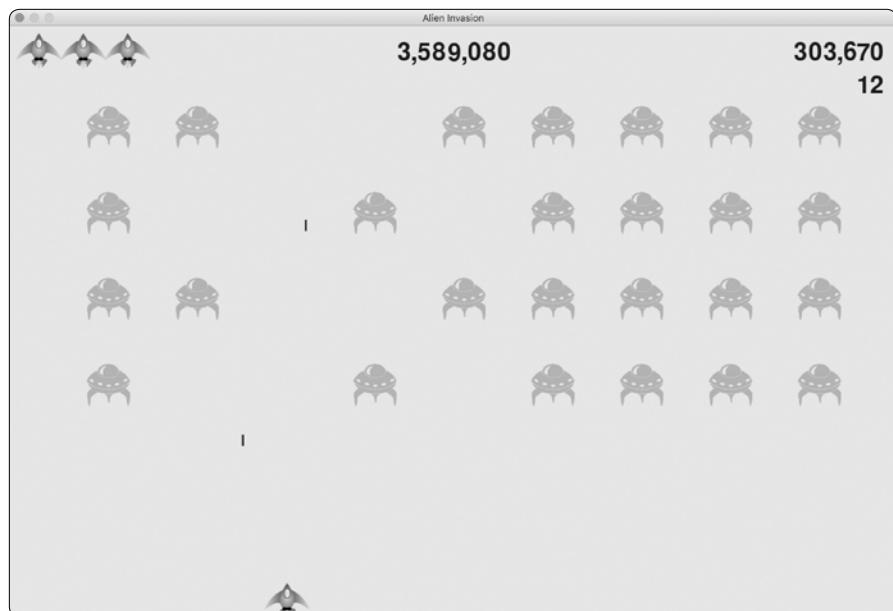


Figure 14-6: The complete scoring system for Alien Invasion

TRY IT YOURSELF

14-5. All-Time High Score: The high score is reset every time a player closes and restarts *Alien Invasion*. Fix this by writing the high score to a file before calling `sys.exit()` and reading in the high score when initializing its value in `GameStats`.

14-6. Refactoring: Look for methods that are doing more than one task, and refactor them to organize your code and make it efficient. For example, move some of the code in `_check_bullet_alien_collisions()`, which starts a new level when the fleet of aliens has been destroyed, to a function called `start_new_level()`. Also, move the four separate method calls in the `__init__()` method in `Scoreboard` to a method called `prep_images()` to shorten `__init__()`. The `prep_images()` method could also help simplify `_check_play_button()` or `start_game()` if you've already refactored `_check_play_button()`.

NOTE Before attempting to refactor the project, see Appendix D to learn how to restore the project to a working state if you introduce bugs while refactoring.

(continued)

14-7. Expanding the Game: Think of a way to expand *Alien Invasion*. For example, you could program the aliens to shoot bullets down at the ship or add shields for your ship to hide behind, which can be destroyed by bullets from either side. Or use something like the `pygame.mixer` module to add sound effects, such as explosions and shooting sounds.

14-8. Sideways Shooter, Final Version: Continue developing *Sideways Shooter*, using everything we've done in this project. Add a Play button, make the game speed up at appropriate points, and develop a scoring system. Be sure to refactor your code as you work, and look for opportunities to customize the game beyond what was shown in this chapter.

Summary

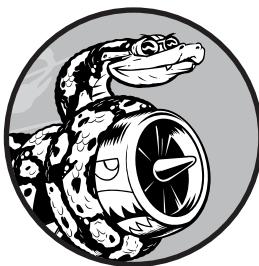
In this chapter, you learned how to implement a Play button to start a new game, detect mouse events, and hide the cursor in active games. You can use what you've learned to create other buttons in your games, like a Help button to display instructions on how to play. You also learned how to modify the speed of a game as it progresses, implement a progressive scoring system, and display information in textual and nontextual ways.

PROJECT 2

DATA VISUALIZATION

15

GENERATING DATA



Data visualization involves exploring data through visual representations. It's closely associated with *data analysis*, which uses code to explore the patterns and connections in a data set. A data set can be made up of a small list of numbers that fits in one line of code or it can be many gigabytes of data.

Making beautiful data representations is about more than pretty pictures. When a representation of a data set is simple and visually appealing, its meaning becomes clear to viewers. People will see patterns and significance in your data sets that they never knew existed.

Fortunately, you don't need a supercomputer to visualize complex data. With Python's efficiency, you can quickly explore data sets made of millions of individual data points on just a laptop. Also, the data points don't have to be numbers. With the basics you learned in the first part of this book, you can analyze nonnumerical data as well.

People use Python for data-intensive work in genetics, climate research, political and economic analysis, and much more. Data scientists have written

an impressive array of visualization and analysis tools in Python, many of which are available to you as well. One of the most popular tools is Matplotlib, a mathematical plotting library. We'll use Matplotlib to make simple plots, such as line graphs and scatter plots. Then we'll create a more interesting data set based on the concept of a random walk—a visualization generated from a series of random decisions.

We'll also use a package called Plotly, which creates visualizations that work well on digital devices. Plotly generates visualizations that automatically resize to fit a variety of display devices. These visualizations can also include a number of interactive features, such as emphasizing particular aspects of the data set when users hover over different parts of the visualization. We'll use Plotly to analyze the results of rolling dice.

Installing Matplotlib

To use Matplotlib for your initial set of visualizations, you'll need to install it using pip, a module that downloads and installs Python packages. Enter the following command at a terminal prompt:

```
$ python -m pip install --user matplotlib
```

This command tells Python to run the pip module and install the matplotlib package to the current user's Python installation. If you use a command other than python on your system to run programs or start a terminal session, such as python3, your command will look like this:

```
$ python3 -m pip install --user matplotlib
```

NOTE

If this command doesn't work on macOS, try running the command again without the --user flag.

To see the kinds of visualizations you can make with Matplotlib, visit the sample gallery at <https://matplotlib.org/gallery/>. When you click a visualization in the gallery, you'll see the code used to generate the plot.

Plotting a Simple Line Graph

Let's plot a simple line graph using Matplotlib, and then customize it to create a more informative data visualization. We'll use the square number sequence 1, 4, 9, 16, 25 as the data for the graph.

Just provide Matplotlib with the numbers, as shown here, and Matplotlib should do the rest:

```
mpl_squares.py
import matplotlib.pyplot as plt
squares = [1, 4, 9, 16, 25]
```

```
❶ fig, ax = plt.subplots()  
ax.plot(squares)  
  
plt.show()
```

We first import the `pyplot` module using the alias `plt` so we don't have to type `pyplot` repeatedly. (You'll see this convention often in online examples, so we'll do the same here.) The `pyplot` module contains a number of functions that generate charts and plots.

We create a list called `squares` to hold the data that we'll plot. Then we follow another common Matplotlib convention by calling the `subplots()` function ❶. This function can generate one or more plots in the same figure. The variable `fig` represents the entire figure or collection of plots that are generated. The variable `ax` represents a single plot in the figure and is the variable we'll use most of the time.

We then use the `plot()` method, which will try to plot the data it's given in a meaningful way. The function `plt.show()` opens Matplotlib's viewer and displays the plot, as shown in Figure 15-1. The viewer allows you to zoom and navigate the plot, and when you click the disk icon, you can save any plot images you like.

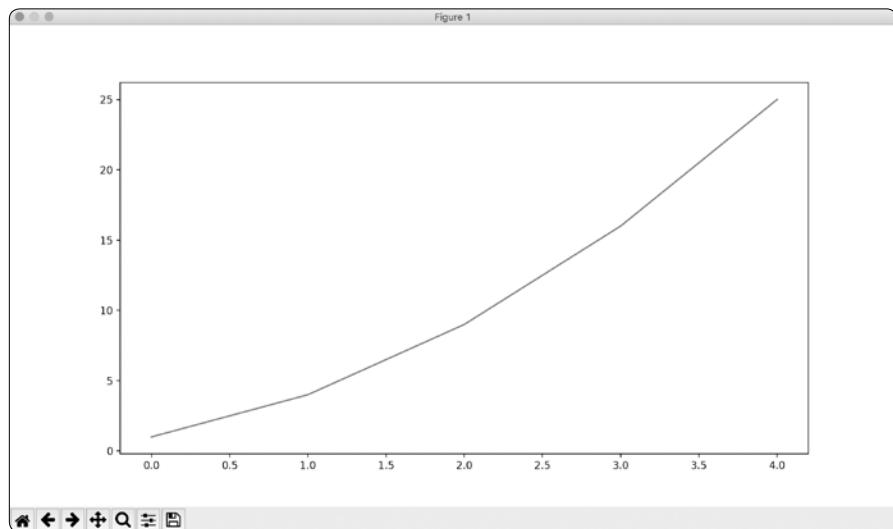


Figure 15-1: One of the simplest plots you can make in Matplotlib

Changing the Label Type and Line Thickness

Although the plot in Figure 15-1 shows that the numbers are increasing, the label type is too small and the line is a little thin to read easily. Fortunately, Matplotlib allows you to adjust every feature of a visualization.

We'll use a few of the available customizations to improve this plot's readability, as shown here:

```
mpl_squares.py import matplotlib.pyplot as plt squares = [1, 4, 9, 16, 25] fig, ax = plt.subplots() ❶ ax.plot(squares, linewidth=3) # Set chart title and label axes. ❷ ax.set_title("Square Numbers", fontsize=24) ❸ ax.set_xlabel("Value", fontsize=14) ax.set_ylabel("Square of Value", fontsize=14) # Set size of tick labels. ❹ ax.tick_params(axis='both', labelsize=14) plt.show()
```

The `linewidth` parameter at ❶ controls the thickness of the line that `plot()` generates. The `set_title()` method at ❷ sets a title for the chart. The `fontsize` parameters, which appear repeatedly throughout the code, control the size of the text in various elements on the chart.

The `set_xlabel()` and `set_ylabel()` methods allow you to set a title for each of the axes ❸, and the method `tick_params()` styles the tick marks ❹. The arguments shown here affect the tick marks on both the x- and y-axes (`axis='both'`) and set the font size of the tick mark labels to 14 (`labelsize=14`).

As you can see in Figure 15-2, the resulting chart is much easier to read. The label type is bigger, and the line graph is thicker. It's often worth experimenting with these values to get an idea of what will look best in the resulting graph.

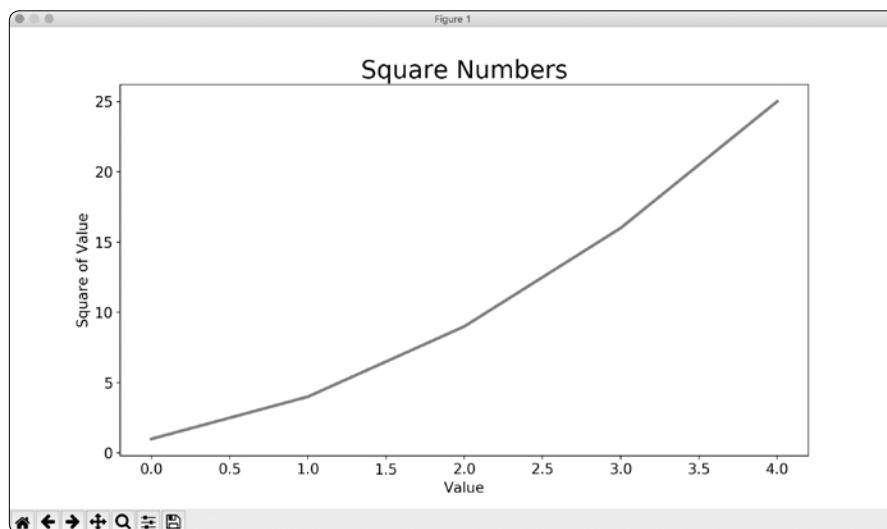


Figure 15-2: The chart is much easier to read now.

Correcting the Plot

But now that we can read the chart better, we see that the data is not plotted correctly. Notice at the end of the graph that the square of 4.0 is shown as 25! Let's fix that.

When you give `plot()` a sequence of numbers, it assumes the first data point corresponds to an x-coordinate value of 0, but our first point corresponds to an x-value of 1. We can override the default behavior by giving `plot()` the input and output values used to calculate the squares:

`mpl_squares.py`

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]

fig, ax = plt.subplots()
ax.plot(input_values, squares, linewidth=3)

# Set chart title and label axes.
--snip--
```

Now `plot()` will graph the data correctly because we've provided the input and output values, so it doesn't have to assume how the output numbers were generated. The resulting plot, shown in Figure 15-3, is correct.

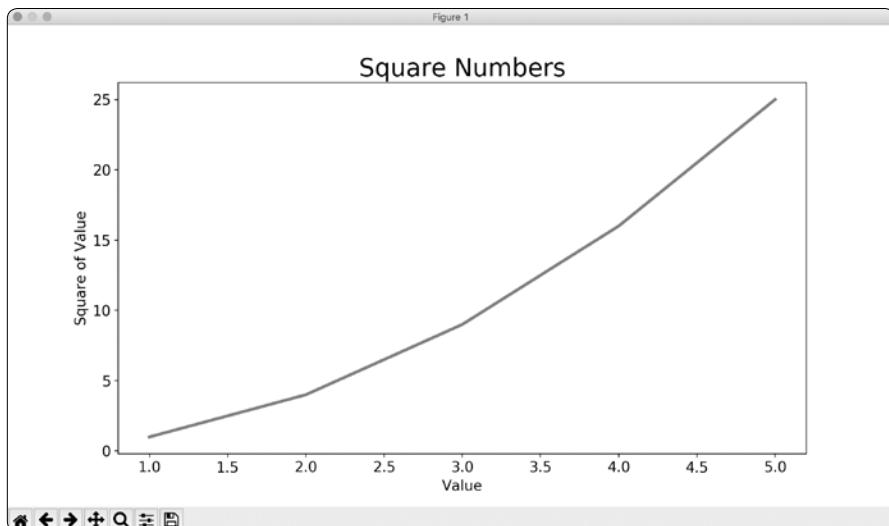


Figure 15-3: The data is now plotted correctly.

You can specify numerous arguments when using `plot()` and use a number of functions to customize your plots. We'll continue to explore these customization functions as we work with more interesting data sets throughout this chapter.

Using Built-in Styles

Matplotlib has a number of predefined styles available, with good starting settings for background colors, gridlines, line widths, fonts, font sizes, and more that will make your visualizations appealing without requiring much customization. To see the styles available on your system, run the following lines in a terminal session:

```
>>> import matplotlib.pyplot as plt  
>>> plt.style.available  
['seaborn-dark', 'seaborn-darkgrid', 'seaborn-ticks', 'fivethirtyeight',  
--snip--
```

To use any of these styles, add one line of code before starting to generate the plot:

```
mpl_squares.py  
  
import matplotlib.pyplot as plt  
  
input_values = [1, 2, 3, 4, 5]  
squares = [1, 4, 9, 16, 25]  
  
plt.style.use('seaborn')  
fig, ax = plt.subplots()  
--snip--
```

This code generates the plot shown in Figure 15-4. A wide variety of styles is available; play around with these styles to find some that you like.

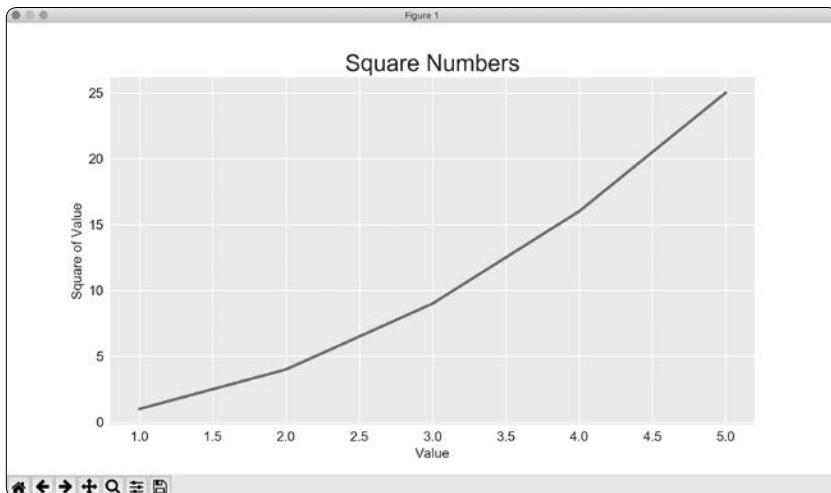


Figure 15-4: The built-in `seaborn` style

Plotting and Styling Individual Points with `scatter()`

Sometimes, it's useful to plot and style individual points based on certain characteristics. For example, you might plot small values in one color and larger values in a different color. You could also plot a large data set with

one set of styling options and then emphasize individual points by replotting them with different options.

To plot a single point, use the `scatter()` method. Pass the single (x, y) values of the point of interest to `scatter()` to plot those values:

scatter_squares.py

```
import matplotlib.pyplot as plt

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(2, 4)

plt.show()
```

Let's style the output to make it more interesting. We'll add a title, label the axes, and make sure all the text is large enough to read:

```
import matplotlib.pyplot as plt

plt.style.use('seaborn')
fig, ax = plt.subplots()
❶ ax.scatter(2, 4, s=200)

# Set chart title and label axes.
ax.set_title("Square Numbers", fontsize=24)
ax.set_xlabel("Value", fontsize=14)
ax.set_ylabel("Square of Value", fontsize=14)

# Set size of tick labels.
ax.tick_params(axis='both', which='major', labelsize=14)

plt.show()
```

At ❶ we call `scatter()` and use the `s` argument to set the size of the dots used to draw the graph. When you run *scatter_squares.py* now, you should see a single point in the middle of the chart, as shown in Figure 15-5.

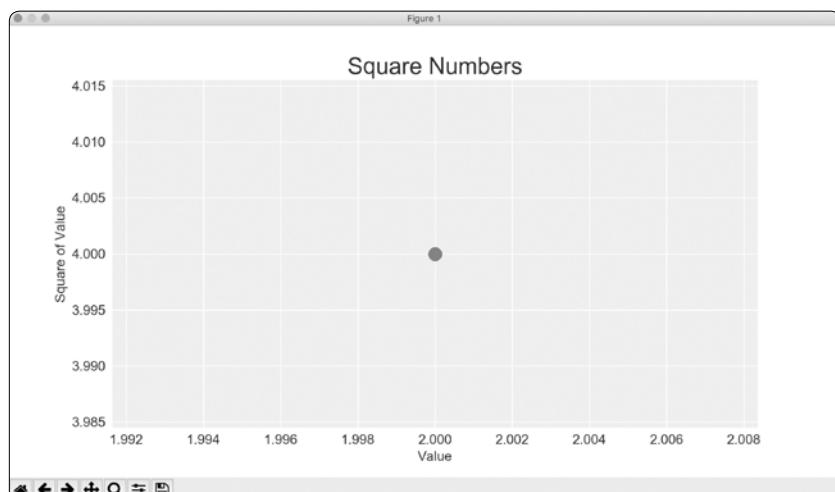


Figure 15-5: Plotting a single point

Plotting a Series of Points with scatter()

To plot a series of points, we can pass `scatter()` separate lists of x- and y-values, like this:

scatter_squares.py

```
import matplotlib.pyplot as plt

x_values = [1, 2, 3, 4, 5]
y_values = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(x_values, y_values, s=100)

# Set chart title and label axes.
--snip--
```

The `x_values` list contains the numbers to be squared, and `y_values` contains the square of each number. When these lists are passed to `scatter()`, Matplotlib reads one value from each list as it plots each point. The points to be plotted are (1, 1), (2, 4), (3, 9), (4, 16), and (5, 25); Figure 15-6 shows the result.

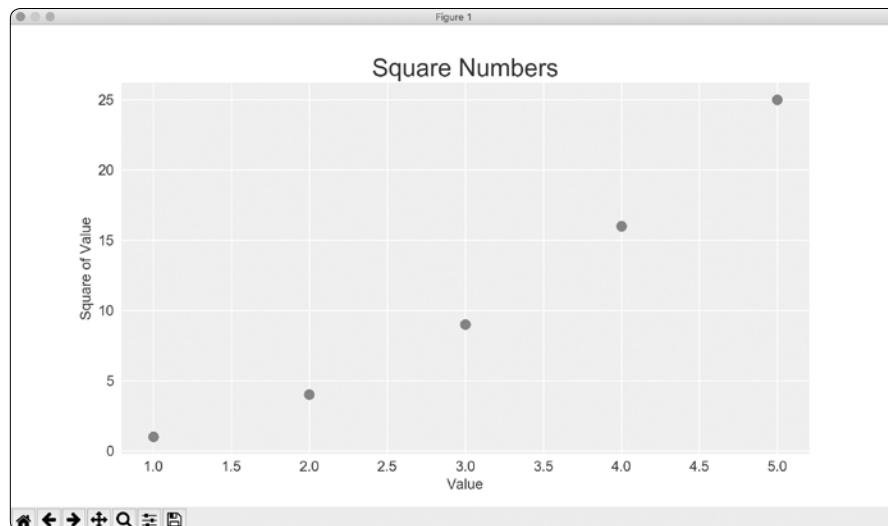


Figure 15-6: A scatter plot with multiple points

Calculating Data Automatically

Writing lists by hand can be inefficient, especially when we have many points. Rather than passing our points in a list, let's use a loop in Python to do the calculations for us.

Here's how this would look with 1000 points:

```
scatter_squares.py    import matplotlib.pyplot as plt

❶ x_values = range(1, 1001)
y_values = [x**2 for x in x_values]

plt.style.use('seaborn')
fig, ax = plt.subplots()
❷ ax.scatter(x_values, y_values, s=10)

# Set chart title and label axes.
--snip--

# Set the range for each axis.
❸ ax.axis([0, 1100, 0, 1100000])

plt.show()
```

We start with a range of x-values containing the numbers 1 through 1000 ❶. Next, a list comprehension generates the y-values by looping through the x-values (for x in x_values), squaring each number ($x**2$) and storing the results in y_values . We then pass the input and output lists to `scatter()` ❷. Because this is a large data set, we use a smaller point size.

At ❸ we use the `axis()` method to specify the range of each axis. The `axis()` method requires four values: the minimum and maximum values for the x-axis and the y-axis. Here, we run the x-axis from 0 to 1100 and the y-axis from 0 to 1,100,000. Figure 15-7 shows the result.

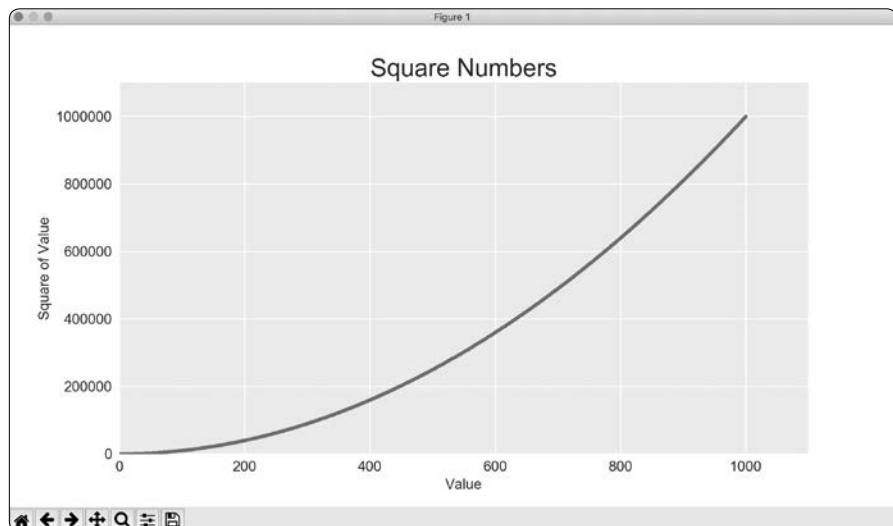


Figure 15-7: Python can plot 1000 points as easily as it plots 5 points.

Defining Custom Colors

To change the color of the points, pass `c` to `scatter()` with the name of a color to use in quotation marks, as shown here:

```
ax.scatter(x_values, y_values, c='red', s=10)
```

You can also define custom colors using the RGB color model. To define a color, pass the `c` argument a tuple with three decimal values (one each for red, green, and blue in that order), using values between 0 and 1. For example, the following line creates a plot with light-green dots:

```
ax.scatter(x_values, y_values, c=(0, 0.8, 0), s=10)
```

Values closer to 0 produce dark colors, and values closer to 1 produce lighter colors.

Using a Colormap

A *colormap* is a series of colors in a gradient that moves from a starting to an ending color. You use colormaps in visualizations to emphasize a pattern in the data. For example, you might make low values a light color and high values a darker color.

The `pyplot` module includes a set of built-in colormaps. To use one of these colormaps, you need to specify how `pyplot` should assign a color to each point in the data set. Here's how to assign each point a color based on its `y`-value:

`scatter_squares.py`

```
import matplotlib.pyplot as plt

x_values = range(1, 1001)
y_values = [x**2 for x in x_values]

ax.scatter(x_values, y_values, c=y_values, cmap=plt.cm.Blues, s=10)

# Set chart title and label axes.
--snip--
```

We pass the list of `y`-values to `c`, and then tell `pyplot` which colormap to use using the `cmap` argument. This code colors the points with lower `y`-values light blue and colors the points with higher `y`-values dark blue. Figure 15-8 shows the resulting plot.

NOTE

You can see all the colormaps available in pyplot at <https://matplotlib.org/>; go to Examples, scroll down to Color, and click Colormap reference.

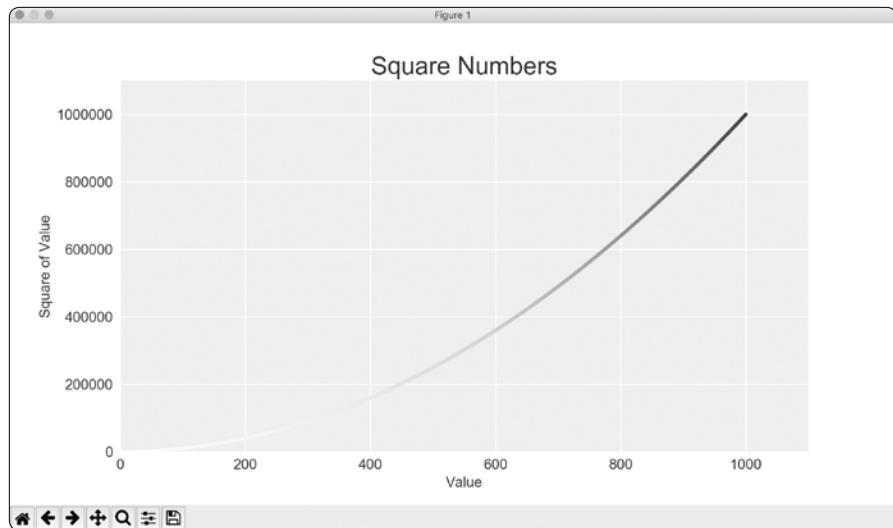


Figure 15-8: A plot using the `Blues` colormap

Saving Your Plots Automatically

If you want your program to automatically save the plot to a file, you can replace the call to `plt.show()` with a call to `plt.savefig()`:

```
plt.savefig('squares_plot.png', bbox_inches='tight')
```

The first argument is a filename for the plot image, which will be saved in the same directory as `scatter_squares.py`. The second argument trims extra whitespace from the plot. If you want the extra whitespace around the plot, just omit this argument.

TRY IT YOURSELF

15-1. Cubes: A number raised to the third power is a *cube*. Plot the first five cubic numbers, and then plot the first 5000 cubic numbers.

15-2. Colored Cubes: Apply a colormap to your cubes plot.

Random Walks

In this section, we'll use Python to generate data for a random walk, and then use Matplotlib to create a visually appealing representation of that data. A *random walk* is a path that has no clear direction but is determined by a series of random decisions, each of which is left entirely to chance. You might imagine a random walk as the path a confused ant would take if it took every step in a random direction.

Random walks have practical applications in nature, physics, biology, chemistry, and economics. For example, a pollen grain floating on a drop of water moves across the surface of the water because it's constantly pushed around by water molecules. Molecular motion in a water drop is random, so the path a pollen grain traces on the surface is a random walk. The code we'll write next models many real-world situations.

Creating the RandomWalk() Class

To create a random walk, we'll create a `RandomWalk` class, which will make random decisions about which direction the walk should take. The class needs three attributes: one variable to store the number of points in the walk and two lists to store the x- and y-coordinate values of each point in the walk.

We'll only need two methods for the `RandomWalk` class: the `__init__()` method and `fill_walk()`, which will calculate the points in the walk. Let's start with `__init__()` as shown here:

```
random_walk.py ❶ from random import choice

class RandomWalk:
    """A class to generate random walks."""

❷     def __init__(self, num_points=5000):
        """Initialize attributes of a walk."""
        self.num_points = num_points

        # All walks start at (0, 0).
❸     self.x_values = [0]
        self.y_values = [0]
```

To make random decisions, we'll store possible moves in a list and use the `choice()` function, from the `random` module, to decide which move to make each time a step is taken ❶. We then set the default number of points in a walk to 5000, which is large enough to generate some interesting patterns but small enough to generate walks quickly ❷. Then at ❸ we make two lists to hold the x- and y-values, and we start each walk at the point (0, 0).

Choosing Directions

We'll use the `fill_walk()` method, as shown here, to fill our walk with points and determine the direction of each step. Add this method to `random_walk.py`:

```
random_walk.py

def fill_walk(self):
    """Calculate all the points in the walk."""

    # Keep taking steps until the walk reaches the desired length.
❶    while len(self.x_values) < self.num_points:

        # Decide which direction to go and how far to go in that direction.
❷        x_direction = choice([1, -1])
```

```

❸     x_distance = choice([0, 1, 2, 3, 4])
❹     x_step = x_direction * x_distance

❺     y_direction = choice([1, -1])
❻     y_distance = choice([0, 1, 2, 3, 4])
❾     y_step = y_direction * y_distance

❿     # Reject moves that go nowhere.
❽     if x_step == 0 and y_step == 0:
❾         continue

❾     # Calculate the new position.
❽     x = self.x_values[-1] + x_step
❽     y = self.y_values[-1] + y_step

❽     self.x_values.append(x)
❽     self.y_values.append(y)

```

At ❶ we set up a loop that runs until the walk is filled with the correct number of points. The main part of the `fill_walk()` method tells Python how to simulate four random decisions: will the walk go right or left? How far will it go in that direction? Will it go up or down? How far will it go in that direction?

We use `choice([1, -1])` to choose a value for `x_direction`, which returns either 1 for right movement or -1 for left ❷. Next, `choice([0, 1, 2, 3, 4])` tells Python how far to move in that direction (`x_distance`) by randomly selecting an integer between 0 and 4. (The inclusion of a 0 allows us to take steps along the `y`-axis as well as steps that have movement along both axes.)

At ❸ and ❹ we determine the length of each step in the `x` and `y` directions by multiplying the direction of movement by the distance chosen. A positive result for `x_step` means move right, a negative result means move left, and 0 means move vertically. A positive result for `y_step` means move up, negative means move down, and 0 means move horizontally. If the value of both `x_step` and `y_step` are 0, the walk doesn't go anywhere, so we continue the loop to ignore this move ❽.

To get the next `x`-value for the walk, we add the value in `x_step` to the last value stored in `x_values` ❽ and do the same for the `y`-values. When we have these values, we append them to `x_values` and `y_values`.

Plotting the Random Walk

Here's the code to plot all the points in the walk:

```

rw_visual.py
import matplotlib.pyplot as plt

from random_walk import RandomWalk

❶ # Make a random walk.
❶ rw = RandomWalk()
❶ rw.fill_walk()

```

```
# Plot the points in the walk.  
plt.style.use('classic')  
fig, ax = plt.subplots()  
❷ ax.scatter(rw.x_values, rw.y_values, s=15)  
plt.show()
```

We begin by importing pyplot and RandomWalk. We then create a random walk and store it in `rw` ❶, making sure to call `fill_walk()`. At ❷ we feed the walk's x- and y-values to `scatter()` and choose an appropriate dot size. Figure 15-9 shows the resulting plot with 5000 points. (The images in this section omit Matplotlib's viewer, but you'll continue to see it when you run `rw_visual.py`.)

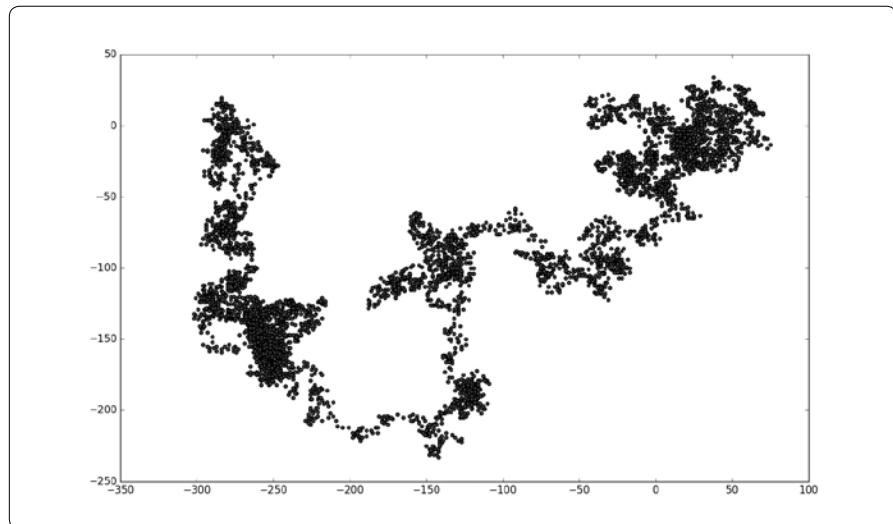


Figure 15-9: A random walk with 5000 points

Generating Multiple Random Walks

Every random walk is different, and it's fun to explore the various patterns that can be generated. One way to use the preceding code to make multiple walks without having to run the program several times is to wrap it in a `while` loop, like this:

```
rw_visual.py  
  
import matplotlib.pyplot as plt  
  
from random_walk import RandomWalk  
  
# Keep making new walks, as long as the program is active.  
while True:  
    # Make a random walk.  
    rw = RandomWalk()  
    rw.fill_walk()  
  
    # Plot the points in the walk.  
    plt.style.use('classic')
```

```
fig, ax = plt.subplots()
ax.scatter(rw.x_values, rw.y_values, s=15)
plt.show()

keep_running = input("Make another walk? (y/n): ")
if keep_running == 'n':
    break
```

This code generates a random walk, displays it in Matplotlib’s viewer, and pauses with the viewer open. When you close the viewer, you’ll be asked whether you want to generate another walk. Press **y** to generate walks that stay near the starting point, that wander off mostly in one direction, that have thin sections connecting larger groups of points, and so on. When you want to end the program, press **n**.

Styling the Walk

In this section, we’ll customize our plots to emphasize the important characteristics of each walk and deemphasize distracting elements. To do so, we identify the characteristics we want to emphasize, such as where the walk began, where it ended, and the path taken. Next, we identify the characteristics to deemphasize, such as tick marks and labels. The result should be a simple visual representation that clearly communicates the path taken in each random walk.

Coloring the Points

We’ll use a colormap to show the order of the points in the walk, and then remove the black outline from each dot so the color of the dots will be clearer. To color the points according to their position in the walk, we pass the `c` argument a list containing the position of each point. Because the points are plotted in order, the list just contains the numbers from 0 to 4999, as shown here:

```
rw_visual.py
--snip--
while True:
    # Make a random walk.
    rw = RandomWalk()
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots()
    ❶    point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
               edgecolors='none', s=15)
    plt.show()

    keep_running = input("Make another walk? (y/n): ")
    --snip--
```

At ❶ we use `range()` to generate a list of numbers equal to the number of points in the walk. Then we store them in the list `point_numbers`, which we'll use to set the color of each point in the walk. We pass `point_numbers` to the `c` argument, use the `Blues` colormap, and then pass `edgecolors='none'` to get rid of the black outline around each point. The result is a plot of the walk that varies from light to dark blue along a gradient, as shown in Figure 15-10.

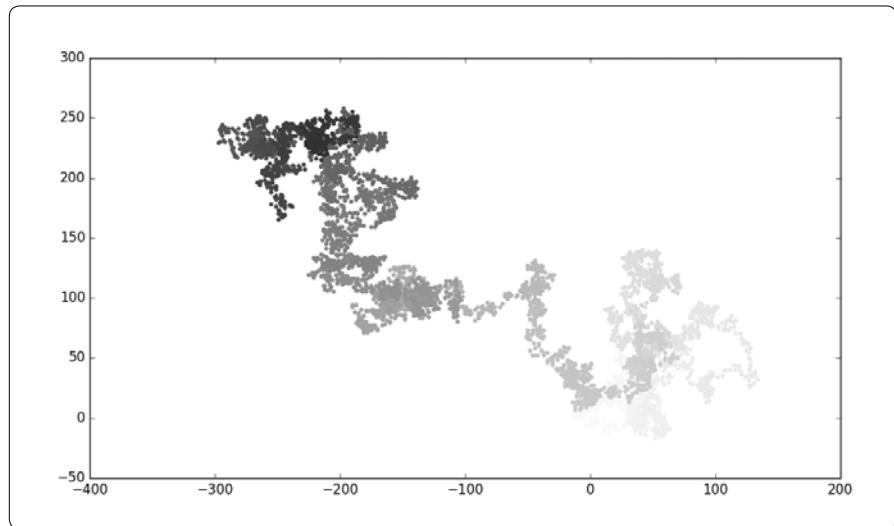


Figure 15-10: A random walk colored with the `Blues` colormap

Plotting the Starting and Ending Points

In addition to coloring points to show their position along the walk, it would be useful to see where each walk begins and ends. To do so, we can plot the first and last points individually after the main series has been plotted. We'll make the end points larger and color them differently to make them stand out, as shown here:

```
rw_visual.py
--snip--
while True:
    --snip--
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
               edgecolors='none', s=15)

    # Emphasize the first and last points.
    ax.scatter(0, 0, c='green', edgecolors='none', s=100)
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
               s=100)

    plt.show()
--snip--
```

To show the starting point, we plot point (0, 0) in green in a larger size (`s=100`) than the rest of the points. To mark the end point, we plot the last x- and y-value in the walk in red with a size of 100. Make sure you insert this code just before the call to `plt.show()` so the starting and ending points are drawn on top of all the other points.

When you run this code, you should be able to spot exactly where each walk begins and ends. (If these end points don't stand out clearly, adjust their color and size until they do.)

Cleaning Up the Axes

Let's remove the axes in this plot so they don't distract from the path of each walk. To turn off the axes, use this code:

```
rw_visual.py
--snip--
while True:
    --snip--
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
               s=100)

    # Remove the axes.
❶    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    plt.show()
--snip--
```

To modify the axes, we use the `ax.get_xaxis()` and `ax.get_yaxis()` methods ❶ to set the visibility of each axis to `False`. As you continue to work with visualizations, you'll frequently see this chaining of methods.

Run `rw_visual.py` now; you should see a series of plots with no axes.

Adding Plot Points

Let's increase the number of points to give us more data to work with. To do so, we increase the value of `num_points` when we make a `RandomWalk` instance and adjust the size of each dot when drawing the plot, as shown here:

```
rw_visual.py
--snip--
while True:
    # Make a random walk.
    rw = RandomWalk(50_000)
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots()
    point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
               edgecolor='none', s=1)
--snip--
```

This example creates a random walk with 50,000 points (to mirror real-world data) and plots each point at size $s=1$. The resulting walk is wispy and cloud-like, as shown in Figure 15-11. As you can see, we've created a piece of art from a simple scatter plot!

Experiment with this code to see how much you can increase the number of points in a walk before your system starts to slow down significantly or the plot loses its visual appeal.

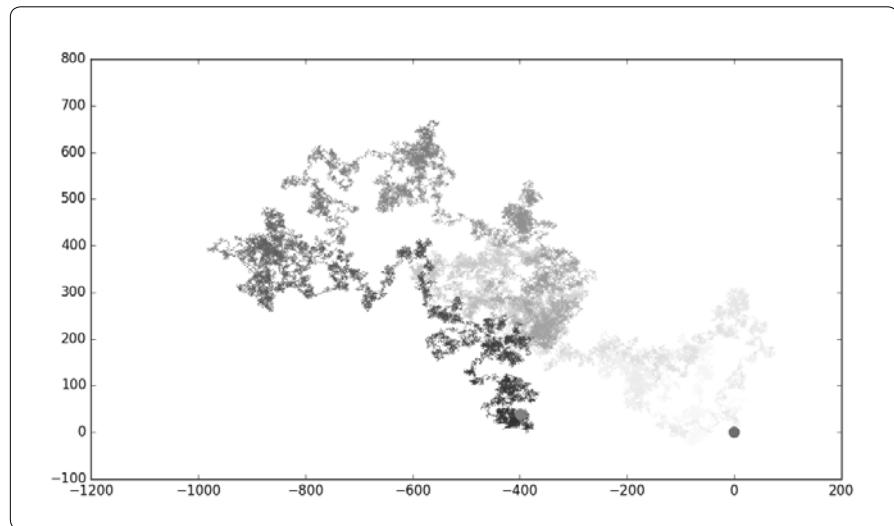


Figure 15-11: A walk with 50,000 points

Altering the Size to Fill the Screen

A visualization is much more effective at communicating patterns in data if it fits nicely on the screen. To make the plotting window better fit your screen, adjust the size of Matplotlib's output, like this:

```
rw_visual.py
--snip--
while True:
    # Make a random walk.
    rw = RandomWalk(50_000)
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots(figsize=(15, 9))
    --snip--
```

When creating the plot, you can pass a `figsize` argument to set the size of the figure. The `figsize` parameter takes a tuple, which tells Matplotlib the dimensions of the plotting window in inches.

Matplotlib assumes that your screen resolution is 100 pixels per inch; if this code doesn't give you an accurate plot size, adjust the numbers as

necessary. Or, if you know your system's resolution, pass `plt.subplots()` the resolution using the `dpi` parameter to set a plot size that makes effective use of the space available on your screen, as shown here:

```
fig, ax = plt.subplots(figsize=(10, 6), dpi=128)
```

TRY IT YOURSELF

15-3. Molecular Motion: Modify `rw_visual.py` by replacing `plt.scatter()` with `plt.plot()`. To simulate the path of a pollen grain on the surface of a drop of water, pass in the `rw.x_values` and `rw.y_values`, and include a `linewidth` argument. Use 5000 instead of 50,000 points.

15-4. Modified Random Walks: In the `RandomWalk` class, `x_step` and `y_step` are generated from the same set of conditions. The direction is chosen randomly from the list `[1, -1]` and the distance from the list `[0, 1, 2, 3, 4]`. Modify the values in these lists to see what happens to the overall shape of your walks. Try a longer list of choices for the distance, such as 0 through 8, or remove the `-1` from the `x` or `y` direction list.

15-5. Refactoring: The `fill_walk()` method is lengthy. Create a new method called `get_step()` to determine the direction and distance for each step, and then calculate the step. You should end up with two calls to `get_step()` in `fill_walk()`:

```
x_step = self.get_step()  
y_step = self.get_step()
```

This refactoring should reduce the size of `fill_walk()` and make the method easier to read and understand.

Rolling Dice with Plotly

In this section, we'll use the Python package Plotly to produce interactive visualizations. Plotly is particularly useful when you're creating visualizations that will be displayed in a browser, because the visualizations will scale automatically to fit the viewer's screen. Visualizations that Plotly generates are also interactive; when the user hovers over certain elements on the screen, information about that element is highlighted.

In this project, we'll analyze the results of rolling dice. When you roll one regular, six-sided die, you have an equal chance of rolling any of the numbers from 1 through 6. However, when you use two dice, you're more likely to roll certain numbers rather than others. We'll try to determine

which numbers are most likely to occur by generating a data set that represents rolling dice. Then we'll plot the results of a large number of rolls to determine which results are more likely than others.

The study of rolling dice is often used in mathematics to explain various types of data analysis. But it also has real-world applications in casinos and other gambling scenarios, as well as in the way games like Monopoly and many role-playing games are played.

Installing Plotly

Install Plotly using pip, just as you did for Matplotlib:

```
$ python -m pip install --user plotly
```

If you used `python3` or something else when installing Matplotlib, make sure you use the same command here.

To see what kind of visualizations are possible with Plotly, visit the gallery of chart types at <https://plot.ly/python/>. Each example includes source code, so you can see how Plotly generates the visualizations.

Creating the Die Class

We'll create the following Die class to simulate the roll of one die:

`die.py`

```
from random import randint

class Die:
    """A class representing a single die."""

❶    def __init__(self, num_sides=6):
        """Assume a six-sided die."""
        self.num_sides = num_sides

    def roll(self):
        """Return a random value between 1 and number of sides."""
❷        return randint(1, self.num_sides)
```

The `__init__()` method takes one optional argument. With the `Die` class, when an instance of our die is created, the number of sides will always be six if no argument is included. If an argument *is* included, that value will set the number of sides on the die ❶. (Dice are named for their number of sides: a six-sided die is a D6, an eight-sided die is a D8, and so on.)

The `roll()` method uses the `randint()` function to return a random number between 1 and the number of sides ❷. This function can return the starting value (1), the ending value (`num_sides`), or any integer between the two.

Rolling the Die

Before creating a visualization based on the `Die` class, let's roll a D6, print the results, and check that the results look reasonable:

```
die_visual.py    from die import Die

# Create a D6.
❶ die = Die()

# Make some rolls, and store results in a list.
results = []
❷ for roll_num in range(100):
    result = die.roll()
    results.append(result)

print(results)
```

At ❶ we create an instance of `Die` with the default six sides. At ❷ we roll the die 100 times and store the results of each roll in the list `results`. Here's a sample set of results:

```
[4, 6, 5, 6, 1, 5, 6, 3, 5, 3, 2, 2, 1, 3, 1, 5, 3, 6, 3, 6, 5, 4,
1, 1, 4, 2, 3, 6, 4, 2, 6, 4, 1, 3, 2, 5, 6, 3, 6, 2, 1, 1, 3, 4, 1, 4,
3, 5, 1, 4, 5, 5, 2, 3, 3, 1, 2, 3, 5, 6, 2, 5, 6, 1, 3, 2, 1, 1, 1, 6,
5, 5, 2, 2, 6, 4, 1, 4, 5, 1, 1, 4, 5, 3, 3, 1, 3, 5, 4, 5, 6, 5, 4,
1, 5, 1, 2]
```

A quick scan of these results shows that the `Die` class seems to be working. We see the values 1 and 6, so we know the smallest and largest possible values are being returned, and because we don't see 0 or 7, we know all the results are in the appropriate range. We also see each number from 1 through 6, which indicates that all possible outcomes are represented. Let's determine exactly how many times each number appears.

Analyzing the Results

We'll analyze the results of rolling one D6 by counting how many times we roll each number:

```
die_visual.py    --snip--
# Make some rolls, and store results in a list.
results = []
❶ for roll_num in range(1000):
    result = die.roll()
    results.append(result)

# Analyze the results.
frequencies = []
❷ for value in range(1, die.num_sides+1):
❸     frequency = results.count(value)
```

```
❸ frequencies.append(frequency)

print(frequencies)
```

Because we're no longer printing the results, we can increase the number of simulated rolls to 1000 ❶. To analyze the rolls, we create the empty list `frequencies` to store the number of times each value is rolled. We loop through the possible values (1 through 6 in this case) at ❷, count how many times each number appears in `results` ❸, and then append this value to the `frequencies` list ❹. We then print this list before making a visualization:

```
[155, 167, 168, 170, 159, 181]
```

These results look reasonable: we see six frequencies, one for each possible number when you roll a D6, and we see that no frequency is significantly higher than any other. Now let's visualize these results.

Making a Histogram

With a list of frequencies, we can make a *histogram* of the results. A histogram is a bar chart showing how often certain results occur. Here's the code to create the histogram:

```
die_visual.py
from plotly.graph_objs import Bar, Layout
from plotly import offline

from die import Die
--snip--

# Analyze the results.
frequencies = []
for value in range(1, die.num_sides+1):
    frequency = results.count(value)
    frequencies.append(frequency)

# Visualize the results.
❶ x_values = list(range(1, die.num_sides+1))
❷ data = [Bar(x=x_values, y=frequencies)]

❸ x_axis_config = {'title': 'Result'}
y_axis_config = {'title': 'Frequency of Result'}
❹ my_layout = Layout(title='Results of rolling one D6 1000 times',
                     xaxis=x_axis_config, yaxis=y_axis_config)
❺ offline.plot({'data': data, 'layout': my_layout}, filename='d6.html')
```

To make a histogram, we need a bar for each of the possible results. We store these in a list called `x_values`, which starts at 1 and ends at the number of sides on the die ❶. Plotly doesn't accept the results of the `range()` function directly, so we need to convert the range to a list explicitly using

the `list()` function. The `Plotly` class `Bar()` represents a data set that will be formatted as a bar chart ❷. This class needs a list of x-values, and a list of y-values. The class must be wrapped in square brackets, because a data set can have multiple elements.

Each axis can be configured in a number of ways, and each configuration option is stored as an entry in a dictionary. At this point, we're just setting the title of each axis ❸. The `Layout()` class returns an object that specifies the layout and configuration of the graph as a whole ❹. Here we set the title of the graph and pass the x- and y-axis configuration dictionaries as well.

To generate the plot, we call the `offline.plot()` function ❺. This function needs a dictionary containing the data and layout objects, and it also accepts a name for the file where the graph will be saved. We store the output in a file called `d6.html`.

When you run the program `die_visual.py`, a browser will probably open showing the file `d6.html`. If this doesn't happen automatically, open a new tab in any web browser, and then open the file `d6.html` (in the folder where you saved `die_visual.py`). You should see a chart that looks like the one in Figure 15-12. (I've modified this chart slightly for printing; by default, `Plotly` generates charts with smaller text than what you see here.)

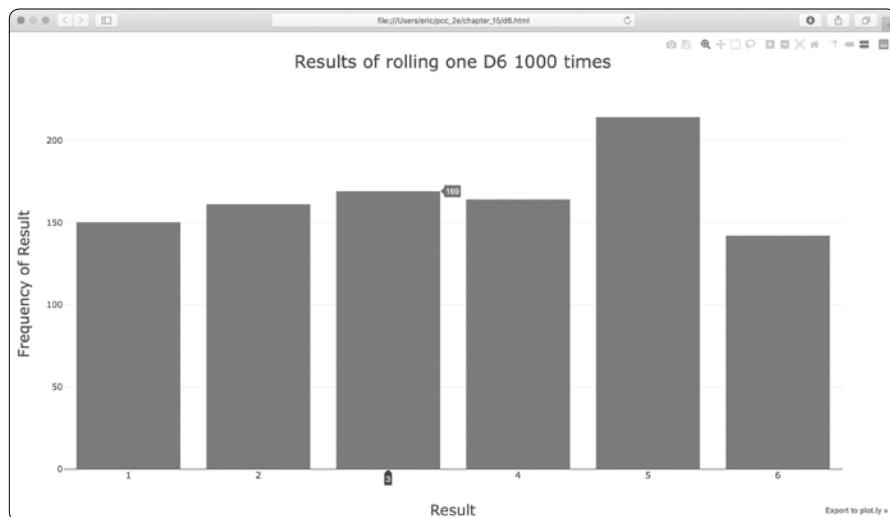


Figure 15-12: A simple bar chart created with Plotly

Notice that `Plotly` has made the chart interactive: hover your cursor over any bar in the chart, and you'll see the associated data. This feature is particularly useful when you're plotting multiple data sets on the same chart. Also notice the icons in the upper right, which allow you to pan and zoom the visualization, and save your visualization as an image.

Rolling Two Dice

Rolling two dice results in larger numbers and a different distribution of results. Let's modify our code to create two D6 dice to simulate the way we roll a pair of dice. Each time we roll the pair, we'll add the two numbers (one from each die) and store the sum in `results`. Save a copy of `die_visual.py` as `dice_visual.py`, and make the following changes:

`dice_visual.py`

```
from plotly.graph_objs import Bar, Layout
from plotly import offline

from die import Die

# Create two D6 dice.
die_1 = Die()
die_2 = Die()

# Make some rolls, and store results in a list.
results = []
for roll_num in range(1000):
    ❶    result = die_1.roll() + die_2.roll()
    results.append(result)

# Analyze the results.
frequencies = []
❷ max_result = die_1.num_sides + die_2.num_sides
❸ for value in range(2, max_result+1):
    frequency = results.count(value)
    frequencies.append(frequency)

# Visualize the results.
x_values = list(range(2, max_result+1))
data = [Bar(x=x_values, y=frequencies)]

❹ x_axis_config = {'title': 'Result', 'dtick': 1}
y_axis_config = {'title': 'Frequency of Result'}
my_layout = Layout(title='Results of rolling two D6 dice 1000 times',
                  xaxis=x_axis_config, yaxis=y_axis_config)
offline.plot({'data': data, 'layout': my_layout}, filename='d6_d6.html')
```

After creating two instances of `Die`, we roll the dice and calculate the sum of the two dice for each roll ❶. The largest possible result (12) is the sum of the largest number on both dice, which we store in `max_result` ❷. The smallest possible result (2) is the sum of the smallest number on both dice. When we analyze the results, we count the number of results for each value between 2 and `max_result` ❸. (We could have used `range(2, 13)`, but this would work only for two D6 dice. When modeling real-world situations, it's best to write code that can easily model a variety of situations. This code allows us to simulate rolling a pair of dice with any number of sides.)

When creating the chart, we include the `dtick` key in the `x_axis_config` dictionary ❹. This setting controls the spacing between tick marks on the x-axis. Now that we have more bars on the histogram, Plotly's default

settings will only label some of the bars. The 'dtick': 1 setting tells Plotly to label every tick mark. We also update the title of the chart and change the output filename as well.

After running this code, you should see a chart that looks like the one in Figure 15-13.

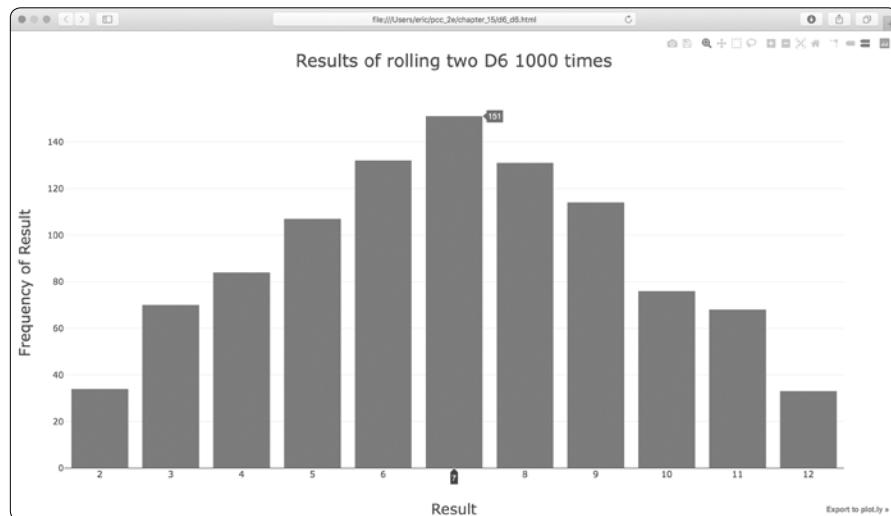


Figure 15-13: Simulated results of rolling two six-sided dice 1000 times

This graph shows the approximate results you're likely to get when you roll a pair of D6 dice. As you can see, you're least likely to roll a 2 or a 12 and most likely to roll a 7. This happens because there are six ways to roll a 7, namely: 1 and 6, 2 and 5, 3 and 4, 4 and 3, 5 and 2, or 6 and 1.

Rolling Dice of Different Sizes

Let's create a six-sided die and a ten-sided die, and see what happens when we roll them 50,000 times:

```
dice_visual.py
from plotly.graph_objs import Bar, Layout
from plotly import offline

from die import Die

# Create a D6 and a D10.
die_1 = Die()
❶ die_2 = Die(10)

# Make some rolls, and store results in a list.
results = []
for roll_num in range(50_000):
    result = die_1.roll() + die_2.roll()
    results.append(result)
```

```

# Analyze the results.
--snip--

# Visualize the results.
x_values = list(range(2, max_result+1))
data = [Bar(x=x_values, y=frequencies)]

x_axis_config = {'title': 'Result', 'dtick': 1}
y_axis_config = {'title': 'Frequency of Result'}
❷ my_layout = Layout(title='Results of rolling a D6 and a D10 50000 times',
                     xaxis=x_axis_config, yaxis=y_axis_config)
offline.plot({'data': data, 'layout': my_layout}, filename='d6_d10.html')

```

To make a D10, we pass the argument `10` when creating the second `Die` instance ❶ and change the first loop to simulate 50,000 rolls instead of 1000. We change the title of the graph and update the output filename as well ❷.

Figure 15-14 shows the resulting chart. Instead of one most likely result, there are five. This happens because there's still only one way to roll the smallest value (1 and 1) and the largest value (6 and 10), but the smaller die limits the number of ways you can generate the middle numbers: there are six ways to roll a 7, 8, 9, 10, and 11. Therefore, these are the most common results, and you're equally likely to roll any one of these numbers.

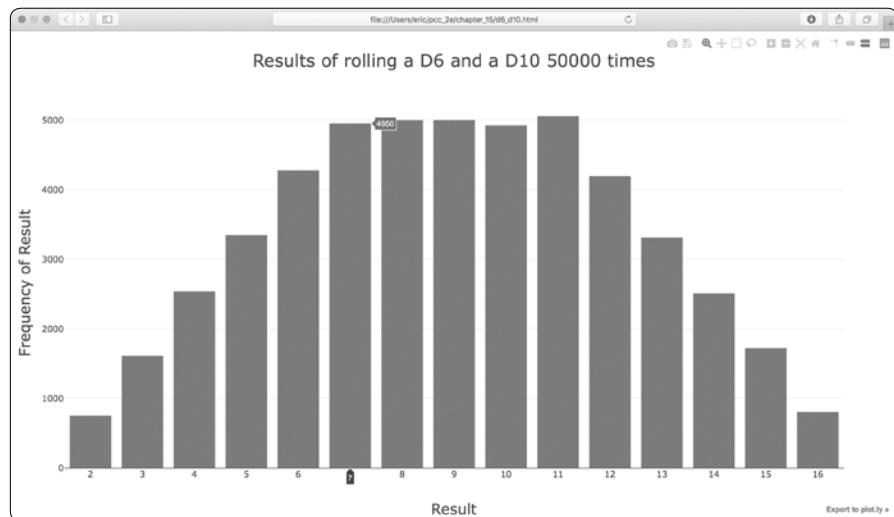


Figure 15-14: The results of rolling a six-sided die and a ten-sided die 50,000 times

Our ability to use Plotly to model the rolling of dice gives us considerable freedom in exploring this phenomenon. In just minutes you can simulate a tremendous number of rolls using a large variety of dice.

TRY IT YOURSELF

15-6. Two D8s: Create a simulation showing what happens when you roll two eight-sided dice 1000 times. Try to picture what you think the visualization will look like before you run the simulation; then see if your intuition was correct. Gradually increase the number of rolls until you start to see the limits of your system’s capabilities.

15-7. Three Dice: When you roll three D6 dice, the smallest number you can roll is 3 and the largest number is 18. Create a visualization that shows what happens when you roll three D6 dice.

15-8. Multiplication: When you roll two dice, you usually add the two numbers together to get the result. Create a visualization that shows what happens if you multiply these numbers instead.

15-9. Die Comprehensions: For clarity, the listings in this section use the long form of `for` loops. If you’re comfortable using list comprehensions, try writing a comprehension for one or both of the loops in each of these programs.

15-10. Practicing with Both Libraries: Try using Matplotlib to make a die-rolling visualization, and use Plotly to make the visualization for a random walk. (You’ll need to consult the documentation for each library to complete this exercise.)

Summary

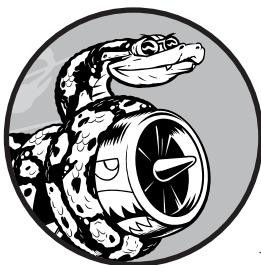
In this chapter, you learned to generate data sets and create visualizations of that data. You created simple plots with Matplotlib and used a scatter plot to explore random walks. You also created a histogram with Plotly and used a histogram to explore the results of rolling dice of different sizes.

Generating your own data sets with code is an interesting and powerful way to model and explore a wide variety of real-world situations. As you continue to work through the data visualization projects that follow, keep an eye out for situations you might be able to model with code. Look at the visualizations you see in news media, and see if you can identify those that were generated using methods similar to the ones you’re learning in these projects.

In Chapter 16, you’ll download data from online sources and continue to use Matplotlib and Plotly to explore that data.

16

DOWNLOADING DATA



In this chapter, you'll download data sets from online sources and create working visualizations of that data. You can find an incredible variety of data online, much of which hasn't been examined thoroughly. The ability to analyze this data allows you to discover patterns and connections that no one else has found.

We'll access and visualize data stored in two common data formats, CSV and JSON. We'll use Python's `csv` module to process weather data stored in the CSV (comma-separated values) format and analyze high and low temperatures over time in two different locations. We'll then use Matplotlib to generate a chart based on our downloaded data to display variations in temperature in two dissimilar environments: Sitka, Alaska, and Death Valley, California. Later in the chapter, we'll use the `json` module to access earthquake data stored in the JSON format and use Plotly to draw a world map showing the locations and magnitudes of recent earthquakes.

By the end of this chapter, you'll be prepared to work with different types and data set formats, and you'll have a deeper understanding of how to build complex visualizations. Being able to access and visualize online data of different types and formats is essential to working with a wide variety of real-world data sets.

The CSV File Format

One simple way to store data in a text file is to write the data as a series of values separated by commas, which is called *comma-separated values*. The resulting files are called *CSV* files. For example, here's a chunk of weather data in CSV format:

```
"USW00025333","SITKA AIRPORT, AK US","2018-01-01","0.45",,"48","38"
```

This is an excerpt of some weather data from January 1, 2018 in Sitka, Alaska. It includes the day's high and low temperatures, as well as a number of other measurements from that day. CSV files can be tricky for humans to read, but they're easy for programs to process and extract values from, which speeds up the data analysis process.

We'll begin with a small set of CSV-formatted weather data recorded in Sitka, which is available in the book's resources at <https://nostarch.com/pythoncrashcourse2e/>. Make a folder called *data* inside the folder where you're saving this chapter's programs. Copy the file *sitka_weather_07-2018_simple.csv* into this new folder. (After you download the book's resources, you'll have all the files you need for this project.)

NOTE

The weather data in this project was originally downloaded from <https://ncdc.noaa.gov/cdo-web/>.

Parsing the CSV File Headers

Python's *csv* module in the standard library parses the lines in a CSV file and allows us to quickly extract the values we're interested in. Let's start by examining the first line of the file, which contains a series of headers for the data. These headers tell us what kind of information the data holds:

```
sitka_highs.py
import csv

filename = 'data/sitka_weather_07-2018_simple.csv'
❶ with open(filename) as f:
❷     reader = csv.reader(f)
❸     header_row = next(reader)
    print(header_row)
```

After importing the *csv* module, we assign the name of the file we're working with to *filename*. We then open the file and assign the resulting file

object to `f` ❶. Next, we call `csv.reader()` and pass it the file object as an argument to create a reader object associated with that file ❷. We assign the reader object to `reader`.

The `csv` module contains a `next()` function, which returns the next line in the file when passed the reader object. In the preceding listing, we call `next()` only once so we get the first line of the file, which contains the file headers ❸. We store the data that's returned in `header_row`. As you can see, `header_row` contains meaningful, weather-related headers that tell us what information each line of data holds:

```
['STATION', 'NAME', 'DATE', 'PRCP', 'TAVG', 'TMAX', 'TMIN']
```

The reader object processes the first line of comma-separated values in the file and stores each as an item in a list. The header `STATION` represents the code for the weather station that recorded this data. The position of this header tells us that the first value in each line will be the weather station code. The `NAME` header indicates that the second value in each line is the name of the weather station that made the recording. The rest of the headers specify what kinds of information were recorded in each reading. The data we're most interested in for now are the date, the high temperature (`TMAX`), and the low temperature (`TMIN`). This is a simple data set that contains only precipitation and temperature-related data. When you download your own weather data, you can choose to include a number of other measurements relating to wind speed, direction, and more detailed precipitation data.

Printing the Headers and Their Positions

To make it easier to understand the file header data, we print each header and its position in the list:

sitka_highs.py

```
--snip--  
with open(filename) as f:  
    reader = csv.reader(f)  
    header_row = next(reader)  
  
❶    for index, column_header in enumerate(header_row):  
        print(index, column_header)
```

The `enumerate()` function returns both the index of each item and the value of each item as you loop through a list ❶. (Note that we've removed the line `print(header_row)` in favor of this more detailed version.)

Here's the output showing the index of each header:

```
0 STATION  
1 NAME  
2 DATE  
3 PRCP  
4 TAVG  
5 TMAX  
6 TMIN
```

Here we see that the dates and their high temperatures are stored in columns 2 and 5. To explore this data, we'll process each row of data in *sitka_weather_07-2018_simple.csv* and extract the values with the indexes 2 and 5.

Extracting and Reading Data

Now that we know which columns of data we need, let's read in some of that data. First, we'll read in the high temperature for each day:

sitka_highs.py

```
--snip--  
with open(filename) as f:  
    reader = csv.reader(f)  
    header_row = next(reader)  
  
    # Get high temperatures from this file.  
    ❶ highs = []  
    ❷ for row in reader:  
        ❸ high = int(row[5])  
        highs.append(high)  
  
print(highs)
```

We make an empty list called `highs` ❶ and then loop through the remaining rows in the file ❷. The `reader` object continues from where it left off in the CSV file and automatically returns each line following its current position. Because we've already read the header row, the loop will begin at the second line where the actual data begins. On each pass through the loop, we pull the data from index 5, which corresponds to the header `TMAX`, and assign it to the variable `high` ❸. We use the `int()` function to convert the data, which is stored as a string, to a numerical format so we can use it. We then append this value to `highs`.

The following listing shows the data now stored in `highs`:

```
[62, 58, 70, 70, 67, 59, 58, 62, 66, 59, 56, 63, 65, 58, 56, 59, 64, 60, 60,  
61, 65, 65, 63, 59, 64, 65, 68, 66, 64, 67, 65]
```

We've extracted the high temperature for each date and stored each value in a list. Now let's create a visualization of this data.

Plotting Data in a Temperature Chart

To visualize the temperature data we have, we'll first create a simple plot of the daily highs using Matplotlib, as shown here:

sitka_highs.py

```
import csv  
  
import matplotlib.pyplot as plt  
  
filename = 'data/sitka_weather_07-2018_simple.csv'  
with open(filename) as f:  
    --snip--
```

```

# Plot the high temperatures.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❶ ax.plot(highs, c='red')

# Format plot.
❷ plt.title("Daily high temperatures, July 2018", fontsize=24)
❸ plt.xlabel('', fontsize=16)
plt.ylabel("Temperature (F)", fontsize=16)
plt.tick_params(axis='both', which='major', labelsize=16)

plt.show()

```

We pass the list of highs to `plot()` and pass `c='red'` to plot the points in red ❶. (We'll plot the highs in red and the lows in blue.) We then specify a few other formatting details, such as the title, font size, and labels ❷, which you should recognize from Chapter 15. Because we have yet to add the dates, we won't label the x-axis, but `plt.xlabel()` does modify the font size to make the default labels more readable ❸. Figure 16-1 shows the resulting plot: a simple line graph of the high temperatures for July 2018 in Sitka, Alaska.

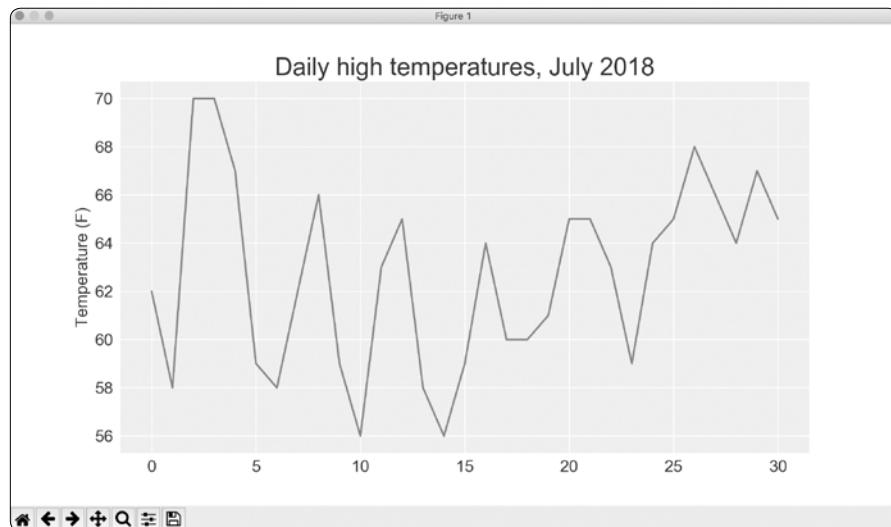


Figure 16-1: A line graph showing daily high temperatures for July 2018 in Sitka, Alaska

The `datetime` Module

Let's add dates to our graph to make it more useful. The first date from the weather data file is in the second row of the file:

```
"USW00025333","SITKA AIRPORT, AK US","2018-07-01","0.25","","62","50"
```

The data will be read in as a string, so we need a way to convert the string "2018-07-01" to an object representing this date. We can construct an object representing July 1, 2018 using the `strptime()` method from the `datetime` module. Let's see how `strptime()` works in a terminal session:

```
>>> from datetime import datetime
>>> first_date = datetime.strptime('2018-07-01', '%Y-%m-%d')
>>> print(first_date)
2018-07-01 00:00:00
```

We first import the `datetime` class from the `datetime` module. Then we call the method `strptime()` using the string containing the date we want to work with as its first argument. The second argument tells Python how the date is formatted. In this example, Python interprets '`%Y-`' to mean the part of the string before the first dash is a four-digit year; '`%m-`' means the part of the string before the second dash is a number representing the month; and '`%d`' means the last part of the string is the day of the month, from 1 to 31.

The `strptime()` method can take a variety of arguments to determine how to interpret the date. Table 16-1 shows some of these arguments.

Table 16-1: Date and Time Formatting Arguments from the `datetime` Module

Argument	Meaning
<code>%A</code>	Weekday name, such as <i>Monday</i>
<code>%B</code>	Month name, such as <i>January</i>
<code>%m</code>	Month, as a number (01 to 12)
<code>%d</code>	Day of the month, as a number (01 to 31)
<code>%Y</code>	Four-digit year, such as 2019
<code>%y</code>	Two-digit year, such as 19
<code>%H</code>	Hour, in 24-hour format (00 to 23)
<code>%I</code>	Hour, in 12-hour format (01 to 12)
<code>%p</code>	AM or PM
<code>%M</code>	Minutes (00 to 59)
<code>%S</code>	Seconds (00 to 61)

Plotting Dates

Now we can improve our temperature data plot by extracting dates for the daily highs and passing those highs and dates to `plot()`, as shown here:

```
sitka_highs.py
import csv
from datetime import datetime

import matplotlib.pyplot as plt

filename = 'data/sitka_weather_07-2018_simple.csv'
```

```

with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    # Get dates and high temperatures from this file.
❶    dates, highs = [], []
    for row in reader:
❷        current_date = datetime.strptime(row[2], '%Y-%m-%d')
        high = int(row[5])
        dates.append(current_date)
        highs.append(high)

    # Plot the high temperatures.
    plt.style.use('seaborn')
    fig, ax = plt.subplots()
❸    ax.plot(dates, highs, c='red')

    # Format plot.
    plt.title("Daily high temperatures, July 2018", fontsize=24)
    plt.xlabel('', fontsize=16)
❹    fig.autofmt_xdate()
    plt.ylabel("Temperature (F)", fontsize=16)
    plt.tick_params(axis='both', which='major', labelsize=16)

plt.show()

```

We create two empty lists to store the dates and high temperatures from the file ❶. We then convert the data containing the date information (`row[2]`) to a `datetime` object ❷ and append it to `dates`. We pass the dates and the high temperature values to `plot()` ❸. The call to `fig.autofmt_xdate()` ❹ draws the date labels diagonally to prevent them from overlapping. Figure 16-2 shows the improved graph.

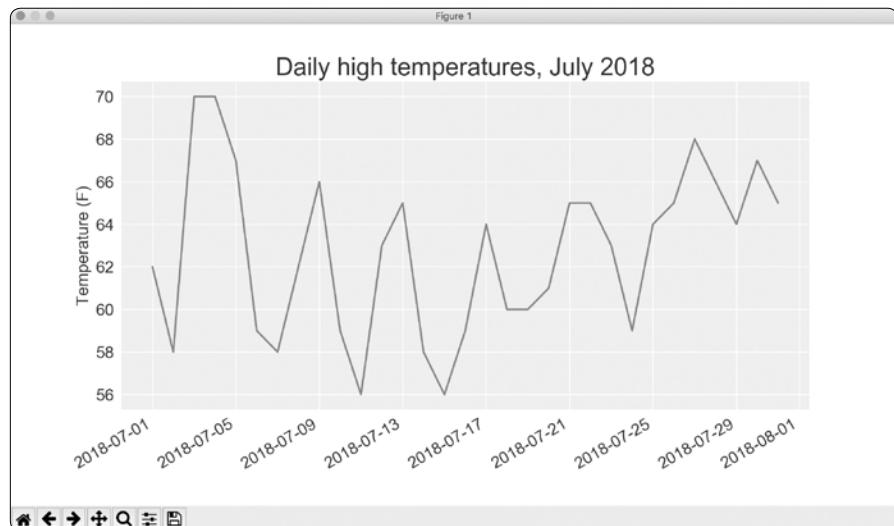


Figure 16-2: The graph is more meaningful now that it has dates on the x-axis.

Plotting a Longer Timeframe

With our graph set up, let's add more data to get a more complete picture of the weather in Sitka. Copy the file *sitka_weather_2018_simple.csv*, which contains a full year's worth of weather data for Sitka, to the folder where you're storing the data for this chapter's programs.

Now we can generate a graph for the entire year's weather:

sitka_highs.py

```
--snip--  
❶ filename = 'data/sitka_weather_2018_simple.csv'  
with open(filename) as f:  
    --snip--  
    # Format plot.  
❷ plt.title("Daily high temperatures - 2018", fontsize=24)  
plt.xlabel('', fontsize=16)  
--snip--
```

We modify the filename to use the new data file *sitka_weather_2018_simple.csv* ❶, and we update the title of our plot to reflect the change in its content ❷. Figure 16-3 shows the resulting plot.

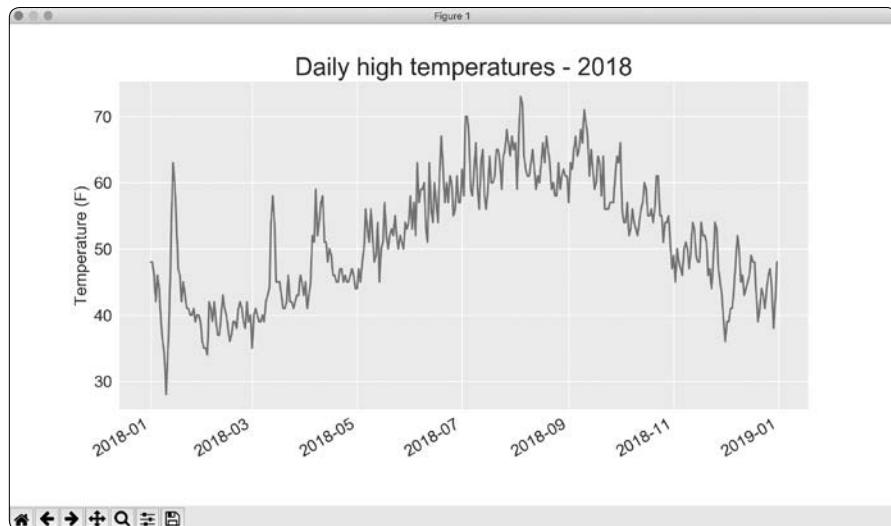


Figure 16-3: A year's worth of data

Plotting a Second Data Series

We can make our informative graph even more useful by including the low temperatures. We need to extract the low temperatures from the data file and then add them to our graph, as shown here:

sitka_highs_lows.py

```
--snip--  
filename = 'sitka_weather_2018_simple.csv'
```

```

with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    # Get dates, and high and low temperatures from this file.
❶    dates, highs, lows = [], [], []
    for row in reader:
        current_date = datetime.strptime(row[2], '%Y-%m-%d')
        high = int(row[5])
❷        low = int(row[6])
        dates.append(current_date)
        highs.append(high)
        lows.append(low)

    # Plot the high and low temperatures.
    plt.style.use('seaborn')
    fig, ax = plt.subplots()
    ax.plot(dates, highs, c='red')
❸    ax.plot(dates, lows, c='blue')

    # Format plot.
❹    plt.title("Daily high and low temperatures - 2018", fontsize=24)
--snip--

```

At ❶ we add the empty list `lows` to hold low temperatures, and then extract and store the low temperature for each date from the seventh position in each row (`row[6]`) ❷. At ❸ we add a call to `plot()` for the low temperatures and color these values blue. Finally, we update the title ❹. Figure 16-4 shows the resulting chart.

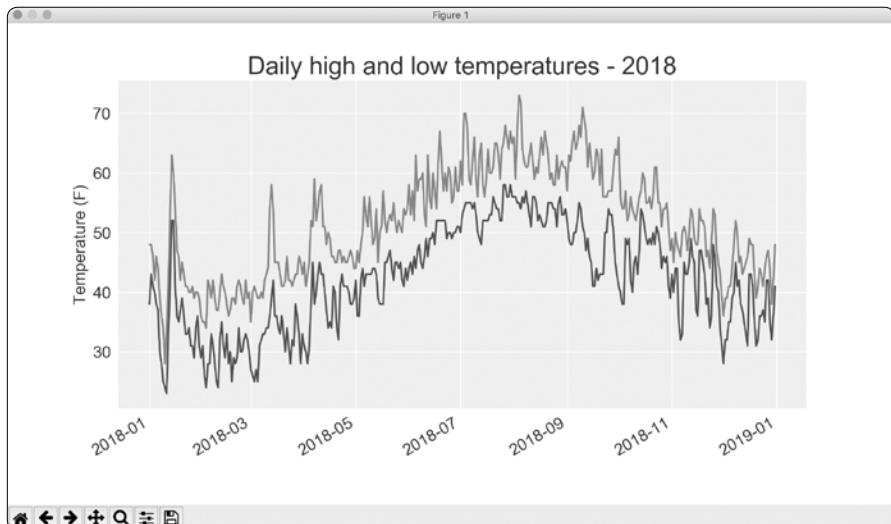


Figure 16-4: Two data series on the same plot

Shading an Area in the Chart

Having added two data series, we can now examine the range of temperatures for each day. Let's add a finishing touch to the graph by using shading to show the range between each day's high and low temperatures. To do so, we'll use the `fill_between()` method, which takes a series of x-values and two series of y-values, and fills the space between the two y-value series:

```
sitka_highs  
_lows.py  
--snip--  
# Plot the high and low temperatures.  
plt.style.use('seaborn')  
fig, ax = plt.subplots()  
❶ ax.plot(dates, highs, c='red', alpha=0.5)  
ax.plot(dates, lows, c='blue', alpha=0.5)  
❷ plt.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1)  
--snip--
```

The `alpha` argument at ❶ controls a color's transparency. An `alpha` value of 0 is completely transparent, and 1 (the default) is completely opaque. By setting `alpha` to 0.5, we make the red and blue plot lines appear lighter.

At ❷ we pass `fill_between()` the list `dates` for the x-values and then the two y-value series `highs` and `lows`. The `facecolor` argument determines the color of the shaded region; we give it a low `alpha` value of 0.1 so the filled region connects the two data series without distracting from the information they represent. Figure 16-5 shows the plot with the shaded region between the highs and lows.

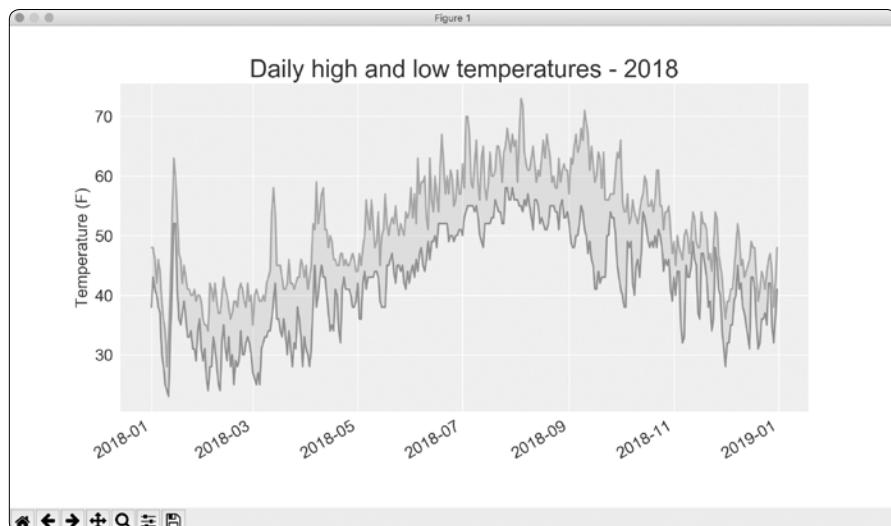


Figure 16-5: The region between the two data sets is shaded.

The shading helps make the range between the two data sets immediately apparent.

Error Checking

We should be able to run the *sitka_highs_lows.py* code using data for any location. But some weather stations collect different data than others, and some occasionally malfunction and fail to collect some of the data they're supposed to. Missing data can result in exceptions that crash our programs unless we handle them properly.

For example, let's see what happens when we attempt to generate a temperature plot for Death Valley, California. Copy the file *death_valley_2018_simple.csv* to the folder where you're storing the data for this chapter's programs.

First, let's run the code to see the headers that are included in this data file:

```
death_valley
_highs_lows.py

import csv

filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    for index, column_header in enumerate(header_row):
        print(index, column_header)
```

Here's the output:

```
0 STATION
1 NAME
2 DATE
3 PRCP
4 TMAX
5 TMIN
6 TOBS
```

The date is in the same position at index 2. But the high and low temperatures are at indexes 4 and 5, so we'd need to change the indexes in our code to reflect these new positions. Instead of including an average temperature reading for the day, this station includes `TOBS`, a reading for a specific observation time.

I removed one of the temperature readings from this file to show what happens when some data is missing from a file. Change *sitka_highs_lows.py* to generate a graph for Death Valley using the indexes we just noted, and see what happens:

```
--snip--
filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    --snip--
    # Get dates, and high and low temperatures from this file.
    dates, highs, lows = [], [], []
    for row in reader:
        current_date = datetime.strptime(row[2], '%Y-%m-%d')
```

```
❶     high = int(row[4])
    low = int(row[5])
    dates.append(current_date)
--snip--
```

At ❶ we update the indexes to correspond to this file's TMAX and TMIN positions.

When we run the program, we get an error, as shown in the last line in the following output:

```
Traceback (most recent call last):
  File "death_valley_highs_lows.py", line 15, in <module>
    high = int(row[4])
ValueError: invalid literal for int() with base 10: ''
```

The traceback tells us that Python can't process the high temperature for one of the dates because it can't turn an empty string ('') into an integer. Rather than look through the data and finding out which reading is missing, we'll just handle cases of missing data directly.

We'll run error-checking code when the values are being read from the CSV file to handle exceptions that might arise. Here's how that works:

```
death_valley
_highs_lows.py
--snip--
filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    --snip--
    for row in reader:
        current_date = datetime.strptime(row[2], '%Y-%m-%d')
   ❶    try:
        high = int(row[4])
        low = int(row[5])
    except ValueError:
   ❷        print(f"Missing data for {current_date}")
   ❸    else:
        dates.append(current_date)
        highs.append(high)
        lows.append(low)

    # Plot the high and low temperatures.
--snip--

    # Format plot.
❹    title = "Daily high and low temperatures - 2018\nDeath Valley, CA"
    plt.title(title, fontsize=20)
    plt.xlabel('', fontsize=16)
--snip--
```

Each time we examine a row, we try to extract the date and the high and low temperature ❶. If any data is missing, Python will raise a `ValueError` and we handle it by printing an error message that includes the date of the missing data ❷. After printing the error, the loop will continue processing the next row. If all data for a date is retrieved without error, the `else` block

will run and the data will be appended to the appropriate lists ❸. Because we're plotting information for a new location, we update the title to include the location on the plot, and we use a smaller font size to accommodate the longer title ❹.

When you run *death_valley_highs_lows.py* now, you'll see that only one date had missing data:

Missing data for 2018-02-18 00:00:00

Because the error is handled appropriately, our code is able to generate a plot, which skips over the missing data. Figure 16-6 shows the resulting plot.

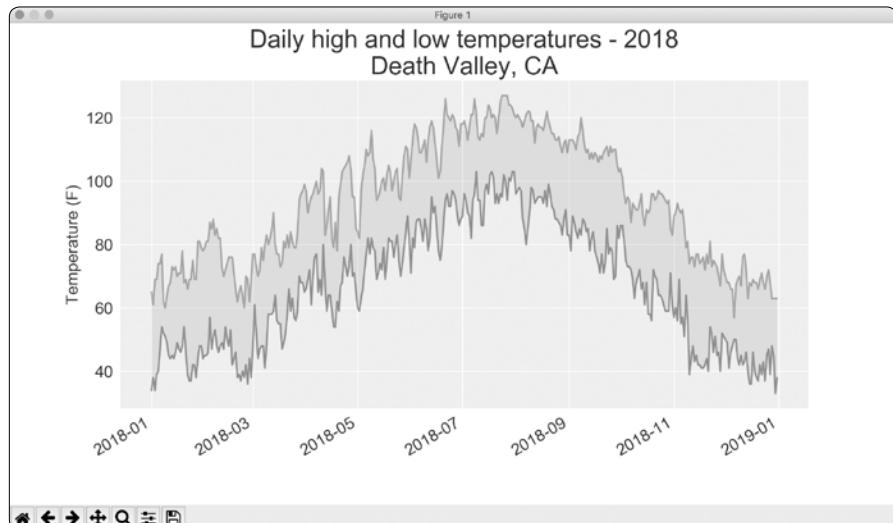


Figure 16-6: Daily high and low temperatures for Death Valley

Comparing this graph to the Sitka graph, we can see that Death Valley is warmer overall than southeast Alaska, as we expect. Also, the range of temperatures each day is greater in the desert. The height of the shaded region makes this clear.

Many data sets you work with will have missing, improperly formatted, or incorrect data. You can use the tools you learned in the first half of this book to handle these situations. Here we used a `try-except-else` block to handle missing data. Sometimes you'll use `continue` to skip over some data or use `remove()` or `del` to eliminate some data after it's been extracted. Use any approach that works, as long as the result is a meaningful, accurate visualization.

Downloading Your Own Data

If you want to download your own weather data, follow these steps:

1. Visit the NOAA Climate Data Online site at <https://www.ncdc.noaa.gov/cdo-web/>. In the *Discover Data By* section, click **Search Tool**. In the *Select a Dataset* box, choose **Daily Summaries**.

2. Select a date range, and in the *Search For* section, choose **ZIP Codes**. Enter the ZIP Code you're interested in, and click **Search**.
3. On the next page, you'll see a map and some information about the area you're focusing on. Below the location name, click **View Full Details**, or click the map and then click **Full Details**.
4. Scroll down and click **Station List** to see the weather stations that are available in this area. Choose one of the stations, and click **Add to Cart**. This data is free, even though the site uses a shopping cart icon. In the upper-right corner, click the cart.
5. In *Select the Output*, choose **Custom GHCN-Daily CSV**. Make sure the date range is correct, and click **Continue**.
6. On the next page, you can select the kinds of data you want. You can download one kind of data, for example, focusing on air temperature, or you can download all the data available from this station. Make your choices, and then click **Continue**.
7. On the last page, you'll see a summary of your order. Enter your email address, and click **Submit Order**. You'll receive a confirmation that your order was received, and in a few minutes you should receive another email with a link to download your data.

The data you download will be structured just like the data we worked with in this section. It might have different headers than those you saw in this section. But if you follow the same steps we used here, you should be able to generate visualizations of the data you're interested in.

TRY IT YOURSELF

16-1. Sitka Rainfall: Sitka is in a temperate rainforest, so it gets a fair amount of rainfall. In the data file *sitka_weather_2018_simple.csv* is a header called PRCP, which represents daily rainfall amounts. Make a visualization focusing on the data in this column. You can repeat the exercise for Death Valley if you're curious how little rainfall occurs in a desert.

16-2. Sitka–Death Valley Comparison: The temperature scales on the Sitka and Death Valley graphs reflect the different data ranges. To accurately compare the temperature range in Sitka to that of Death Valley, you need identical scales on the y-axis. Change the settings for the y-axis on one or both of the charts in Figures 16-5 and 16-6. Then make a direct comparison between temperature ranges in Sitka and Death Valley (or any two places you want to compare).

16-3. San Francisco: Are temperatures in San Francisco more like temperatures in Sitka or temperatures in Death Valley? Download some data for San Francisco, and generate a high-low temperature plot for San Francisco to make a comparison.

16-4. Automatic Indexes: In this section, we hardcoded the indexes corresponding to the TMIN and TMAX columns. Use the header row to determine the indexes for these values, so your program can work for Sitka or Death Valley. Use the station name to automatically generate an appropriate title for your graph as well.

16-5. Explore: Generate a few more visualizations that examine any other weather aspect you're interested in for any locations you're curious about.

Mapping Global Data Sets: JSON Format

In this section, you'll download a data set representing all the earthquakes that have occurred in the world during the previous month. Then you'll make a map showing the location of these earthquakes and how significant each one was. Because the data is stored in the JSON format, we'll work with it using the `json` module. Using Plotly's beginner-friendly mapping tool for location-based data, you'll create visualizations that clearly show the global distribution of earthquakes.

Downloading Earthquake Data

Copy the file `eq_1_day_m1.json` to the folder where you're storing the data for this chapter's programs. Earthquakes are categorized by their magnitude on the Richter scale. This file includes data for all earthquakes with a magnitude M1 or greater that took place in the last 24 hours (at the time of this writing). This data comes from one of the United States Geological Survey's earthquake data feeds, which you can find at <https://earthquake.usgs.gov/earthquakes/feed/>.

Examining JSON Data

When you open `eq_1_day_m1.json`, you'll see that it's very dense and hard to read:

```
{"type": "FeatureCollection", "metadata": {"generated": 1550361461000, ...  
{"type": "Feature", "properties": {"mag": 1.2, "place": "11km NNE of Nor...  
{"type": "Feature", "properties": {"mag": 4.3, "place": "69km NNW of Ayn...  
{"type": "Feature", "properties": {"mag": 3.6, "place": "126km SSE of Co...  
{"type": "Feature", "properties": {"mag": 2.1, "place": "21km NNW of Teh...  
{"type": "Feature", "properties": {"mag": 4, "place": "57km SSW of Kakto...  
--snip--
```

This file is formatted more for machines than it is for humans. But we can see that the file contains some dictionaries, as well as information that we're interested in, such as earthquake magnitudes and locations.

The `json` module provides a variety of tools for exploring and working with JSON data. Some of these tools will help us reformat the file so we can look at the raw data more easily before we begin to work with it programmatically.

Let's start by loading the data and displaying it in a format that's easier to read. This is a long data file, so instead of printing it, we'll rewrite the data to a new file. Then we can open that file and scroll back and forth easily through the data:

```
eq_explore
_data.py
import json
# Explore the structure of the data.
filename = 'data/eq_data_1_day_m1.json'
with open(filename) as f:
❶    all_eq_data = json.load(f)
❷    readable_file = 'data/readable_eq_data.json'
    with open(readable_file, 'w') as f:
❸        json.dump(all_eq_data, f, indent=4)
```

We first import the `json` module to load the data properly from the file, and then store the entire set of data in `all_eq_data` ❶. The `json.load()` function converts the data into a format Python can work with: in this case, a giant dictionary. At ❷ we create a file to write this same data into a more readable format. The `json.dump()` function takes a JSON data object and a file object, and writes the data to that file ❸. The `indent=4` argument tells `dump()` to format the data using indentation that matches the data's structure.

When you look in your `data` directory and open the file `readable_eq_data.json`, here's the first part of what you'll see:

```
{          "type": "FeatureCollection",
❶    "metadata": {
        "generated": 1550361461000,
        "url": "https://earthquake.usgs.gov/earthquakes/.../1.0_day.geojson",
        "title": "USGS Magnitude 1.0+ Earthquakes, Past Day",
        "status": 200,
        "api": "1.7.0",
        "count": 158
    },
❷    "features": [
--snip--
```

The first part of the file includes a section with the key "`metadata`". This tells us when the data file was generated and where we can find the data online. It also gives us a human-readable title and the number of earthquakes included in this file. In this 24-hour period, 158 earthquakes were recorded.

This geoJSON file has a structure that's helpful for location-based data. The information is stored in a list associated with the key "features" ❷. Because this file contains earthquake data, the data is in list form where every item in the list corresponds to a single earthquake. This structure might look confusing, but it's quite powerful. It allows geologists to store as much information as they need to in a dictionary about each earthquake, and then stuff all those dictionaries into one big list.

Let's look at a dictionary representing a single earthquake:

```
readable_eq
_data.json
--snip--
{
    "type": "Feature",
❶    "properties": {
        "mag": 0.96,
        --snip--
❷        "title": "M 1.0 - 8km NE of Aguanga, CA"
    },
❸    "geometry": {
        "type": "Point",
        "coordinates": [
            -116.7941667,
            33.4863333,
            3.22
        ]
    },
    "id": "ci37532978"
},
```

The key "properties" contains a lot of information about each earthquake ❶. We're mainly interested in the magnitude of each quake, which is associated with the key "mag". We're also interested in the title of each earthquake, which provides a nice summary of its magnitude and location ❷.

The key "geometry" helps us understand where the earthquake occurred ❸. We'll need this information to map each event. We can find the longitude ❹ and the latitude ❺ for each earthquake in a list associated with the key "coordinates".

This file contains way more nesting than we'd use in the code we write, so if it looks confusing, don't worry: Python will handle most of the complexity. We'll only be working with one or two nesting levels at a time. We'll start by pulling out a dictionary for each earthquake that was recorded in the 24-hour time period.

NOTE

When we talk about locations, we often say the location's latitude first, followed by the longitude. This convention probably arose because humans discovered latitude long before we developed the concept of longitude. However, many geospatial frameworks list the longitude first and then the latitude, because this corresponds to the (x, y) convention we use in mathematical representations. The geoJSON format follows the (longitude, latitude) convention, and if you use a different framework it's important to learn what convention that framework follows.

Making a List of All Earthquakes

First, we'll make a list that contains all the information about every earthquake that occurred.

```
eq_explore
_data.py

import json

# Explore the structure of the data.
filename = 'data/eq_data_1_day_m1.json'
with open(filename) as f:
    all_eq_data = json.load(f)

all_eq_dicts = all_eq_data['features']
print(len(all_eq_dicts))
```

We take the data associated with the key 'features' and store it in `all_eq_dicts`. We know this file contains records about 158 earthquakes, and the output verifies that we've captured all of the earthquakes in the file:

158

Notice how short this code is. The neatly formatted file `readable_eq_data.json` has over 6,000 lines. But in just a few lines, we can read through all that data and store it in a Python list. Next, we'll pull the magnitudes from each earthquake.

Extracting Magnitudes

Using the list containing data about each earthquake, we can loop through that list and extract any information we want. Now we'll pull the magnitude of each earthquake:

```
--snip--
all_eq_dicts = all_eq_data['features']

❶ mags = []
for eq_dict in all_eq_dicts:
❷     mag = eq_dict['properties']['mag']
    mags.append(mag)

print(mags[:10])
```

We make an empty list to store the magnitudes, and then loop through the dictionary `all_eq_dicts` ❶. Inside this loop, each earthquake is represented by the dictionary `eq_dict`. Each earthquake's magnitude is stored in the 'properties' section of this dictionary under the key 'mag' ❷. We store each magnitude in the variable `mag`, and then append it to the list `mags`.

We print the first 10 magnitudes, so we can see whether we're getting the correct data:

```
[0.96, 1.2, 4.3, 3.6, 2.1, 4, 1.06, 2.3, 4.9, 1.8]
```

Next, we'll pull the location data for each earthquake, and then we can make a map of the earthquakes.

Extracting Location Data

The location data is stored under the key "geometry". Inside the geometry dictionary is a "coordinates" key, and the first two values in this list are the longitude and latitude. Here's how we'll pull this data:

```
eq_explore
_data.py
--snip--
all_eq_dicts = all_eq_data['features']

mags, lons, lats = [], [], []
for eq_dict in all_eq_dicts:
    mag = eq_dict['properties']['mag']
❶    lon = eq_dict['geometry']['coordinates'][0]
    lat = eq_dict['geometry']['coordinates'][1]
    mags.append(mag)
    lons.append(lon)
    lats.append(lat)

print(mags[:10])
print(lons[:5])
print(lats[:5])
```

We make empty lists for the longitudes and latitudes. The code `eq_dict['geometry']` accesses the dictionary representing the geometry element of the earthquake ❶. The second key, 'coordinates', pulls the list of values associated with 'coordinates'. Finally, the 0 index asks for the first value in the list of coordinates, which corresponds to an earthquake's longitude.

When we print the first five longitudes and latitudes, the output shows that we're pulling the correct data:

```
[0.96, 1.2, 4.3, 3.6, 2.1, 4, 1.06, 2.3, 4.9, 1.8]
[-116.7941667, -148.9865, -74.2343, -161.6801, -118.5316667]
[33.4863333, 64.6673, -12.1025, 54.2232, 35.3098333]
```

With this data, we can move on to mapping each earthquake.

Building a World Map

With the information we've pulled so far, we can build a simple world map. Although it won't look presentable yet, we want to make sure the information is displayed correctly before focusing on style and presentation issues. Here's the initial map:

```
eq_world_map.py
import json

❶ from plotly.graph_objs import Scattergeo, Layout
from plotly import offline

--snip--
```

```

for eq_dict in all_eq_dicts:
    --snip--

    # Map the earthquakes.
② data = [Scattergeo(lon=lons, lat=lats)]
③ my_layout = Layout(title='Global Earthquakes')

④ fig = {'data': data, 'layout': my_layout}
offline.plot(fig, filename='global_earthquakes.html')

```

We import the `Scattergeo` chart type and the `Layout` class, and then import the `offline` module to render the map ❶. As we did when making a bar chart, we define a list called `data`. We create the `Scattergeo` object inside this list ❷, because you can plot more than one data set on any visualization you make. A `Scattergeo` chart type allows you to overlay a scatter plot of geographic data on a map. In the simplest use of this chart type, you only need to provide a list of longitudes and a list of latitudes.

We give the chart an appropriate title ❸ and create a dictionary called `fig` that contains the data and the layout ❹. Finally, we pass `fig` to the `plot()` function along with a descriptive filename for the output. When you run this file, you should see a map that looks like the one in Figure 16-7. Earthquakes usually occur near plate boundaries, which matches what we see in the chart.

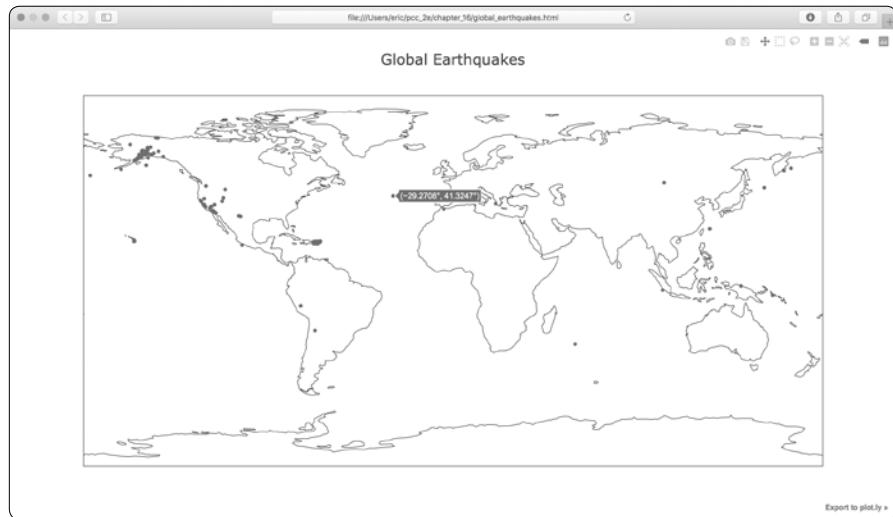


Figure 16-7: A simple map showing where all the earthquakes in the last 24 hours occurred

We can do a lot of modifications to make this map more meaningful and easier to read, so let's make some of these changes.

A Different Way of Specifying Chart Data

Before we configure the chart, let's look at a slightly different way to specify the data for a Plotly chart. In the current chart, the data list is defined in one line:

```
data = [Scattergeo(lon=lons, lat=lats)]
```

This is one of the simplest ways to define the data for a chart in Plotly. But it's not the best way when you want to customize the presentation. Here's an equivalent way to define the data for the current chart:

```
data = [{  
    'type': 'scattergeo',  
    'lon': lons,  
    'lat': lats,  
}]
```

In this approach, all the information about the data is structured as key-value pairs in a dictionary. If you put this code into *eq_plot.py*, you'll see the same chart we just generated. This format allows us to specify customizations more easily than the previous format.

Customizing Marker Size

When we're figuring out how to improve the map's styling, we should focus on aspects of the data that we want to communicate more clearly. The current map shows the location of each earthquake, but it doesn't communicate the severity of any earthquake. We want viewers to immediately see where the most significant earthquakes occur in the world.

To do this, we'll change the size of markers depending on the magnitude of each earthquake:

```
eq_world_map.py  
import json  
--snip--  
# Map the earthquakes.  
data = [{  
    'type': 'scattergeo',  
    'lon': lons,  
    'lat': lats,  
❶    'marker': {  
❷        'size': [5*mag for mag in mags],  
    },  
}]  
my_layout = Layout(title='Global Earthquakes')  
--snip--
```

Plotly offers a huge variety of customizations you can make to a data series, each of which can be expressed as a key-value pair. Here we're using the key 'marker' to specify how big each marker on the map should be ❶. We use a nested dictionary as the value associated with 'marker', because you can specify a number of settings for all the markers in a series.

We want the size to correspond to the magnitude of each earthquake. But if we just pass in the `mags` list, the markers would be too small to easily see the size differences. We need to multiply the magnitude by a scale factor to get an appropriate marker size. On my screen, a value of 5 works well; a slightly smaller or larger value might work better for your map. We use a list comprehension, which generates an appropriate marker size for each value in the `mags` list ❷.

When you run this code, you should see a map that looks like the one in Figure 16-8. This is a much better map, but we can still do more.

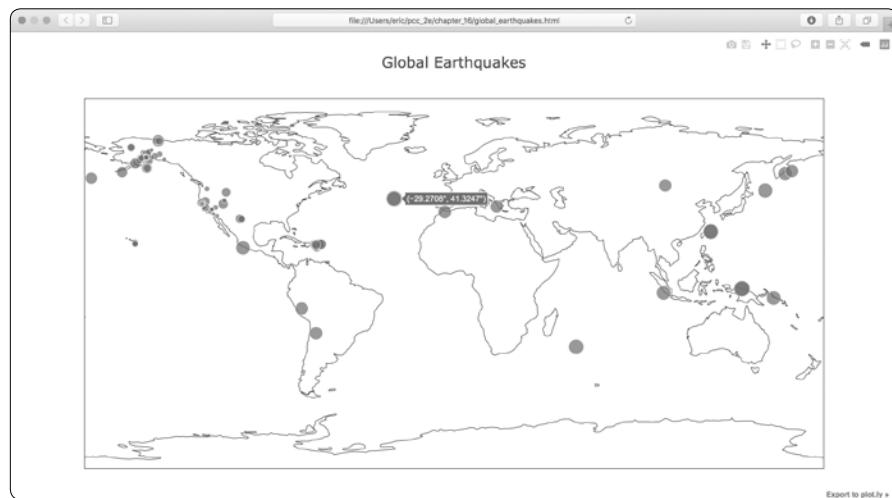


Figure 16-8: The map now shows the magnitude of each earthquake.

Customizing Marker Colors

We can also customize each marker's color to provide some classification to the severity of each earthquake. We'll use Plotly's colorscales to do this. Before you make these changes, copy the file `eq_data_30_day_m1.json` to your data directory. This file includes earthquake data for a 30-day period, and the map will be much more interesting to look at using this larger data set.

Here's how to use a colorscale to represent the magnitude of each earthquake:

```
eq_world_map.py
--snip--
❶ filename = 'data/eq_data_30_day_m1.json'
--snip--
# Map the earthquakes.
data = [
    --snip--
```

```

        'marker': {
            'size': [5*mag for mag in mags],
②            'color': mags,
③            'colorscale': 'Viridis',
④            'reversescale': True,
⑤            'colorbar': {'title': 'Magnitude'},
        },
    }]
--snip--

```

Be sure to update the filename so you're using the 30-day data set ❶. All the significant changes here occur in the 'marker' dictionary, because we're only modifying the markers' appearance. The 'color' setting tells Plotly what values it should use to determine where each marker falls on the colorscale ❷. We use the `mags` list to determine the color that's used. The 'colorscale' setting tells Plotly which range of colors to use: 'Viridis' is a colorscale that ranges from dark blue to bright yellow and works well for this data set ❸. We set 'reversescale' to True, because we want to use bright yellow for the lowest values and dark blue for the most severe earthquakes ❹. The 'colorbar' setting allows us to control the appearance of the colorscale shown on the side of the map. Here we title the colorscale 'Magnitude' to make it clear what the colors represent ❺.

When you run the program now, you'll see a much nicer-looking map. In Figure 16-9, the colorscale shows the severity of individual earthquakes. Plotting this many earthquakes really makes it clear where the tectonic plate boundaries are!

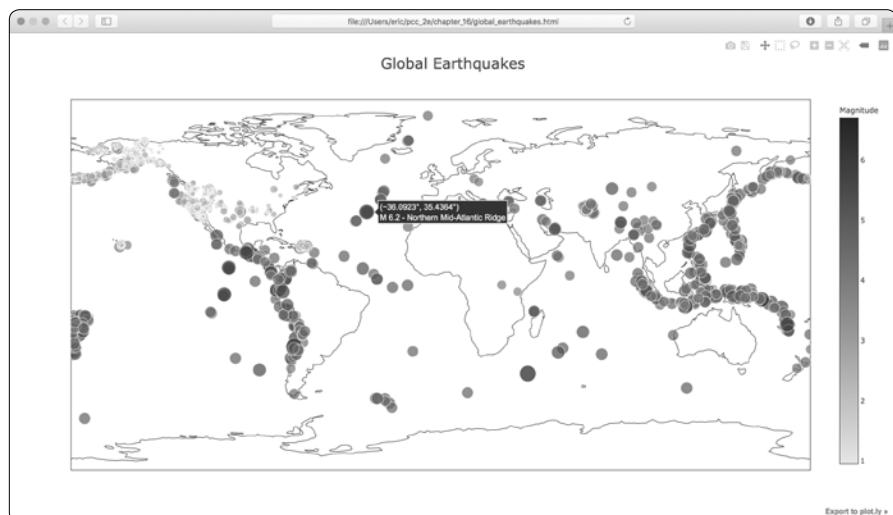


Figure 16-9: In 30 days' worth of earthquakes, color and size are used to represent the magnitude of each earthquake.

Other Colorscales

You can also choose from a number of other colorscales. To see the available colorscales, save the following short program as `show_color_scales.py`:

```
show_color_scales.py
from plotly import colors
for key in colors.PLOTLY_SCALES.keys():
    print(key)
```

Plotly stores the colorscales in the `colors` module. The colorscales are defined in the dictionary `PLOTLY_SCALES`, and the names of the colorscales serve as the keys in the dictionary. Here's the output showing all the available colorscales:

```
Greys
YlGnBu
Greens
--snip--
Viridis
```

Feel free to try out these colorscales; remember that you can reverse any of these scales using the `reversescale` setting.

NOTE

If you print the `PLOTLY_SCALES` dictionary, you can see how colorscales are defined. Every scale has a beginning color and an end color, and some scales have one or more intermediate colors defined as well. Plotly interpolates shades between each of these defined colors.

Adding Hover Text

To finish this map, we'll add some informative text that appears when you hover over the marker representing an earthquake. In addition to showing the longitude and latitude, which appear by default, we'll show the magnitude and provide a description of the approximate location as well.

To make this change, we need to pull a little more data from the file and add it to the dictionary in `data` as well:

```
eq_world_map.py
--snip--
❶ mags, lons, lats, hover_texts = [], [], [], []
for eq_dict in all_eq_dicts:
    --snip--
    lat = eq_dict['geometry']['coordinates'][1]
    title = eq_dict['properties']['title']
    mags.append(mag)
    lons.append(lon)
    lats.append(lat)
    hover_texts.append(title)
--snip--
```

```

# Map the earthquakes.
data = [
    'type': 'scattergeo',
    'lon': lons,
    'lat': lats,
❸  'text': hover_texts,
    'marker': {
        '--snip--'
    },
}]
--snip--

```

We first make a list called `hover_texts` to store the label we'll use for each marker ❶. The “title” section of the earthquake data contains a descriptive name of the magnitude and location of each earthquake in addition to its longitude and latitude. At ❷ we pull this information and assign it to the variable `title`, and then append it to the list `hover_texts`.

When we include the key `'text'` in the `data` object, Plotly uses this value as a label for each marker when the viewer hovers over the marker. When we pass a list that matches the number of markers, Plotly pulls an individual label for each marker it generates ❸. When you run this program, you should be able to hover over any marker, see a description of where that earthquake took place, and read its exact magnitude.

This is impressive! In approximately 40 lines of code, we've created a visually appealing and meaningful map of global earthquake activity that also illustrates the geological structure of the planet. Plotly offers a wide range of ways you can customize the appearance and behavior of your visualizations. Using Plotly's many options, you can make charts and maps that show exactly what you want them to.

TRY IT YOURSELF

16-6. Refactoring: The loop that pulls data from `all_eq_dicts` uses variables for the magnitude, longitude, latitude, and title of each earthquake before appending these values to their appropriate lists. This approach was chosen for clarity in how to pull data from a JSON file, but it's not necessary in your code. Instead of using these temporary variables, pull each value from `eq_dict` and append it to the appropriate list in one line. Doing so should shorten the body of this loop to just four lines.

16-7. Automated Title: In this section, we specified the title manually when defining `my_layout`, which means we have to remember to update the title every time the source file changes. Instead, you can use the title for the data set in the metadata part of the JSON file. Pull this value, assign it to a variable, and use this for the title of the map when you're defining `my_layout`.

(continued)

16-8. Recent Earthquakes: You can find data files containing information about the most recent earthquakes over 1-hour, 1-day, 7-day, and 30-day periods online. Go to <https://earthquake.usgs.gov/earthquakes/feed/v1.0/geojson.php> and you'll see a list of links to data sets for various time periods, focusing on earthquakes of different magnitudes. Download one of these data sets, and create a visualization of the most recent earthquake activity.

16-9. World Fires: In the resources for this chapter, you'll find a file called `world_fires_1_day.csv`. This file contains information about fires burning in different locations around the globe, including the latitude and longitude, and the brightness of each fire. Using the data processing work from the first part of this chapter and the mapping work from this section, make a map that shows which parts of the world are affected by fires.

You can download more recent versions of this data at <https://earthdata.nasa.gov/earth-observation-data/near-real-time/firms/active-fire-data/>. You can find links to the data in CSV format in the *TXT* section.

Summary

In this chapter, you learned to work with real-world data sets. You processed CSV and JSON files, and extracted the data you want to focus on. Using historical weather data, you learned more about working with Matplotlib, including how to use the `datetime` module and how to plot multiple data series on one chart. You plotted geographical data on a world map in Plotly and styled Plotly maps and charts as well.

As you gain experience working with CSV and JSON files, you'll be able to process almost any data you want to analyze. You can download most online data sets in either or both of these formats. By working with these formats, you'll be able to learn how to work with other data formats more easily as well.

In the next chapter, you'll write programs that automatically gather their own data from online sources, and then you'll create visualizations of that data. These are fun skills to have if you want to program as a hobby and critical skills if you're interested in programming professionally.

17

WORKING WITH APIs



In this chapter, you'll learn how to write a self-contained program that generates a visualization based on data that it retrieves. Your program will use a web *application programming interface (API)* to automatically request specific information from a website—rather than entire pages—and then use that information to generate a visualization. Because programs written like this will always use current data to generate a visualization, even when that data might be rapidly changing, it will always be up to date.

Using a Web API

A web API is a part of a website designed to interact with programs. Those programs use very specific URLs to request certain information. This kind of request is called an *API call*. The requested data will be returned in an

easily processed format, such as JSON or CSV. Most apps that rely on external data sources, such as apps that integrate with social media sites, rely on API calls.

Git and GitHub

We'll base our visualization on information from GitHub, a site that allows programmers to collaborate on coding projects. We'll use GitHub's API to request information about Python projects on the site, and then generate an interactive visualization of the relative popularity of these projects using Plotly.

GitHub (<https://github.com/>) takes its name from Git, a distributed version control system. Git helps people manage their work on a project, so changes made by one person won't interfere with changes other people are making. When you implement a new feature in a project, Git tracks the changes you make to each file. When your new code works, you *commit* the changes you've made, and Git records the new state of your project. If you make a mistake and want to revert your changes, you can easily return to any previously working state. (To learn more about version control using Git, see Appendix D.) Projects on GitHub are stored in *repositories*, which contain everything associated with the project: its code, information on its collaborators, any issues or bug reports, and so on.

When users on GitHub like a project, they can "star" it to show their support and keep track of projects they might want to use. In this chapter, we'll write a program to automatically download information about the most-starred Python projects on GitHub, and then we'll create an informative visualization of these projects.

Requesting Data Using an API Call

GitHub's API lets you request a wide range of information through API calls. To see what an API call looks like, enter the following into your browser's address bar and press ENTER:

`https://api.github.com/search/repositories?q=language:python&sort=stars`

This call returns the number of Python projects currently hosted on GitHub, as well as information about the most popular Python repositories. Let's examine the call. The first part, <https://api.github.com/>, directs the request to the part of GitHub that responds to API calls. The next part, `search/repositories`, tells the API to conduct a search through all repositories on GitHub.

The question mark after `repositories` signals that we're about to pass an argument. The `q` stands for *query*, and the equal sign (=) lets us begin specifying a query (`q=`). By using `language:python`, we indicate that we want information only on repositories that have Python as the primary language. The final part, `&sort=stars`, sorts the projects by the number of stars they've been given.

The following snippet shows the first few lines of the response.

```
{  
❶ "total_count": 3494012,  
❷ "incomplete_results": false,  
❸ "items": [  
  {  
    "id": 21289110,  
    "node_id": "MDEwOlJlcG9zaXRvcnkyMTI40TExMA==",  
    "name": "awesome-python",  
    "full_name": "vinta/awesome-python",  
    --snip--
```

You can see from the response that this URL is not primarily intended to be entered by humans, because it's in a format that's meant to be processed by a program. GitHub found 3,494,012 Python projects as of this writing ❶. Because the value for "incomplete_results" is false, we know that the request was successful (it's not incomplete) ❷. If GitHub had been unable to fully process the API request, it would have returned true here. The "items" returned are displayed in the list that follows, which contains details about the most popular Python projects on GitHub ❸.

Installing Requests

The Requests package allows a Python program to easily request information from a website and examine the response. Use pip to install Requests:

```
$ python -m pip install --user requests
```

This line tells Python to run the pip module and install the Requests package to the current user's Python installation. If you use `python3` or a different command when running programs or installing packages, make sure you use the same command here.

NOTE

If this command doesn't work on macOS, try running the command again without the --user flag.

Processing an API Response

Now we'll begin to write a program to automatically issue an API call and process the results by identifying the most starred Python projects on GitHub:

```
python_repos.py ❶ import requests  
  
    # Make an API call and store the response.  
❷ url = 'https://api.github.com/search/repositories?q=language:python&sort=stars'  
❸ headers = {'Accept': 'application/vnd.github.v3+json'}  
❹ r = requests.get(url, headers=headers)  
❺ print(f"Status code: {r.status_code}")
```

```
# Store API response in a variable.  
❶ response_dict = r.json()  
  
# Process results.  
print(response_dict.keys())
```

At ❶ we import the `requests` module. At ❷ we store the URL of the API call in the `url` variable. GitHub is currently on the third version of its API, so we define headers for the API call ❸ that ask explicitly to use this version of the API. Then we use `requests` to make the call to the API ❹.

We call `get()` and pass it the URL and the header that we defined, and we assign the response object to the variable `r`. The response object has an attribute called `status_code`, which tells us whether the request was successful. (A status code of 200 indicates a successful response.) At ❺ we print the value of `status_code` so we can make sure the call went through successfully.

The API returns the information in JSON format, so we use the `json()` method to convert the information to a Python dictionary ❻. We store the resulting dictionary in `response_dict`.

Finally, we print the keys from `response_dict` and see this output:

```
Status code: 200  
dict_keys(['total_count', 'incomplete_results', 'items'])
```

Because the status code is 200, we know that the request was successful. The response dictionary contains only three keys: `'total_count'`, `'incomplete_results'`, and `'items'`. Let's take a look inside the response dictionary.

NOTE

Simple calls like this should return a complete set of results, so it's safe to ignore the value associated with `'incomplete_results'`. But when you're making more complex API calls, your program should check this value.

Working with the Response Dictionary

With the information from the API call stored as a dictionary, we can work with the data stored there. Let's generate some output that summarizes the information. This is a good way to make sure we received the information we expected and to start examining the information we're interested in:

`python_repos.py`

```
import requests  
  
# Make an API call and store the response.  
--snip--  
  
# Store API response in a variable.  
response_dict = r.json()  
❶ print(f"Total repositories: {response_dict['total_count']}")  
  
# Explore information about the repositories.  
❷ repo_dicts = response_dict['items']  
print(f"Repositories returned: {len(repo_dicts)}")
```

```
# Examine the first repository.  
❸ repo_dict = repo_dicts[0]  
❹ print(f"\nKeys: {len(repo_dict)}")  
❺ for key in sorted(repo_dict.keys()):  
    print(key)
```

At ❶ we print the value associated with 'total_count', which represents the total number of Python repositories on GitHub.

The value associated with 'items' is a list containing a number of dictionaries, each of which contains data about an individual Python repository. At ❷ we store this list of dictionaries in `repo_dicts`. We then print the length of `repo_dicts` to see how many repositories we have information for.

To look closer at the information returned about each repository, we pull out the first item from `repo_dicts` and store it in `repo_dict` ❸. We then print the number of keys in the dictionary to see how much information we have ❹. At ❺ we print all the dictionary's keys to see what kind of information is included.

The results give us a clearer picture of the actual data:

```
Status code: 200  
Total repositories: 3494030  
Repositories returned: 30
```

```
❶ Keys: 73  
archive_url  
archived  
assignees_url  
--snip--  
url  
watchers  
watchers_count
```

GitHub's API returns a lot of information about each repository: there are 73 keys in `repo_dict` ❶. When you look through these keys, you'll get a sense of the kind of information you can extract about a project. (The only way to know what information is available through an API is to read the documentation or to examine the information through code, as we're doing here.)

Let's pull out the values for some of the keys in `repo_dict`:

```
python_repos.py  
--snip--  
# Explore information about the repositories.  
repo_dicts = response_dict['items']  
print(f"Repositories returned: {len(repo_dicts)}")  
  
# Examine the first repository.  
repo_dict = repo_dicts[0]  
  
print("\nSelected information about first repository:")  
❶ print(f"Name: {repo_dict['name']}")  
❷ print(f"Owner: {repo_dict['owner']['login']}")  
❸ print(f"Stars: {repo_dict['stargazers_count']}")  
print(f"Repository: {repo_dict['html_url']}")
```

```
❸ print(f"Created: {repo_dict['created_at']}")  
❹ print(f"Updated: {repo_dict['updated_at']}")  
❺ print(f"Description: {repo_dict['description']}")
```

Here, we print the values for a number of keys from the first repository’s dictionary. At ❶ we print the name of the project. An entire dictionary represents the project’s owner, so at ❷ we use the key `owner` to access the dictionary representing the owner, and then use the key `login` to get the owner’s login name. At ❸ we print how many stars the project has earned and the URL for the project’s GitHub repository. We then show when it was created ❹ and when it was last updated ❺. Finally, we print the repository’s description; the output should look something like this:

```
Status code: 200  
Total repositories: 3494032  
Repositories returned: 30  
  
Selected information about first repository:  
Name: awesome-python  
Owner: vinta  
Stars: 61549  
Repository: https://github.com/vinta/awesome-python  
Created: 2014-06-27T21:00:06Z  
Updated: 2019-02-17T04:30:00Z  
Description: A curated list of awesome Python frameworks, libraries, software  
and resources
```

We can see that the most-starred Python project on GitHub as of this writing is `awesome-python`, its owner is user `vinta`, and it has been starred by more than 60,000 GitHub users. We can see the URL for the project’s repository, its creation date of June 2014, and that it was updated recently. Additionally, the description tells us that `awesome-python` contains a list of popular Python resources.

Summarizing the Top Repositories

When we make a visualization for this data, we’ll want to include more than one repository. Let’s write a loop to print selected information about each repository the API call returns so we can include them all in the visualization:

```
python_repos.py  
--snip--  
# Explore information about the repositories.  
repo_dicts = response_dict['items']  
print(f"Repositories returned: {len(repo_dicts)}")  
  
❶ print("\nSelected information about each repository:")  
❷ for repo_dict in repo_dicts:  
    print(f"\nName: {repo_dict['name']}")  
    print(f"Owner: {repo_dict['owner']['login']}")  
    print(f"Stars: {repo_dict['stargazers_count']}")  
    print(f"Repository: {repo_dict['html_url']}")  
    print(f"Description: {repo_dict['description']}")
```

We print an introductory message at ❶. At ❷ we loop through all the dictionaries in `repo_dicts`. Inside the loop, we print the name of each project, its owner, how many stars it has, its URL on GitHub, and the project’s description, as shown here:

```
Status code: 200
Total repositories: 3494040
Repositories returned: 30
```

Selected information about each repository:

```
Name: awesome-python
Owner: vinta
Stars: 61549
Repository: https://github.com/vinta/awesome-python
Description: A curated list of awesome Python frameworks, libraries, software
and resources

Name: system-design-primer
Owner: donnemartin
Stars: 57256
Repository: https://github.com/donnemartin/system-design-primer
Description: Learn how to design large-scale systems. Prep for the system
design interview. Includes Anki flashcards.
--snip--

Name: python-patterns
Owner: faif
Stars: 19058
Repository: https://github.com/faif/python-patterns
Description: A collection of design patterns/idioms in Python
```

Some interesting projects appear in these results, and it might be worth looking at a few. But don’t spend too much time, because shortly we’ll create a visualization that will make the results much easier to read.

Monitoring API Rate Limits

Most APIs are rate limited, which means there’s a limit to how many requests you can make in a certain amount of time. To see if you’re approaching GitHub’s limits, enter https://api.github.com/rate_limit into a web browser. You should see a response that begins like this:

```
{
  "resources": {
    "core": {
      "limit": 60,
      "remaining": 58,
      "reset": 1550385312
    },
❶    "search": {
❷      "limit": 10,
❸      "remaining": 8,
```

```
❸     "reset": 1550381772
},
--snip--
```

The information we're interested in is the rate limit for the search API ❶. We see at ❷ that the limit is 10 requests per minute and that we have 8 requests remaining for the current minute ❸. The reset value represents the time in *Unix* or *epoch time* (the number of seconds since midnight on January 1, 1970) when our quota will reset ❹. If you reach your quota, you'll get a short response that lets you know you've reached the API limit. If you reach the limit, just wait until your quota resets.

NOTE

Many APIs require you to register and obtain an API key to make API calls. As of this writing, GitHub has no such requirement, but if you obtain an API key, your limits will be much higher.

Visualizing Repositories Using Plotly

Let's make a visualization using the data we have now to show the relative popularity of Python projects on GitHub. We'll make an interactive bar chart: the height of each bar will represent the number of stars the project has acquired, and you can click the bar's label to go to that project's home on GitHub. Save a copy of the program we've been working on as *python_repos_visual.py*, and then modify it so it reads as follows:

python_repos_visual.py

```
import requests

❶ from plotly.graph_objs import Bar
from plotly import offline

❷ # Make an API call and store the response.
url = 'https://api.github.com/search/repositories?q=language:python&sort=stars'
headers = {'Accept': 'application/vnd.github.v3+json'}
r = requests.get(url, headers=headers)
print(f"Status code: {r.status_code}")

# Process results.
response_dict = r.json()
repo_dicts = response_dict['items']
❸ repo_names, stars = [], []
for repo_dict in repo_dicts:
    repo_names.append(repo_dict['name'])
    stars.append(repo_dict['stargazers_count'])

# Make visualization.
❹ data = [{
    'type': 'bar',
    'x': repo_names,
    'y': stars,
}]
```

```

❸ my_layout = {
    'title': 'Most-Starred Python Projects on GitHub',
    'xaxis': {'title': 'Repository'},
    'yaxis': {'title': 'Stars'},
}

fig = {'data': data, 'layout': my_layout}
offline.plot(fig, filename='python_repos.html')

```

We import the `Bar` class and the `offline` module from `plotly` ❶. We don't need to import the `Layout` class because we'll use the dictionary approach to define the layout, just as we did for the `data` list in the earthquake mapping project in Chapter 16. We continue to print the status of the API call response so we'll know if there is a problem ❷. We also remove some of the code that processes the API response, because we're no longer in the exploratory phase; we know we have the data we want.

We then create two empty lists ❸ to store the data we'll include in the initial chart. We'll need the name of each project to label the bars, and the number of stars to determine the height of the bars. In the loop, we append the name of each project and the number of stars it has to these lists.

Next, we define the `data` list ❹. This contains a dictionary, like we used in Chapter 16, which defines the type of the plot and provides the data for the x- and y-values. The x-values are the names of the projects, and the y-values are the number of stars each project has been given.

At ❺ we define the layout for this chart using the dictionary approach. Instead of making an instance of the `Layout` class, we build a dictionary with the layout specifications we want to use. We set a title for the overall chart, and we define a label for each axis.

Figure 17-1 shows the resulting chart. We can see that the first few projects are significantly more popular than the rest, but all of them are important projects in the Python ecosystem.

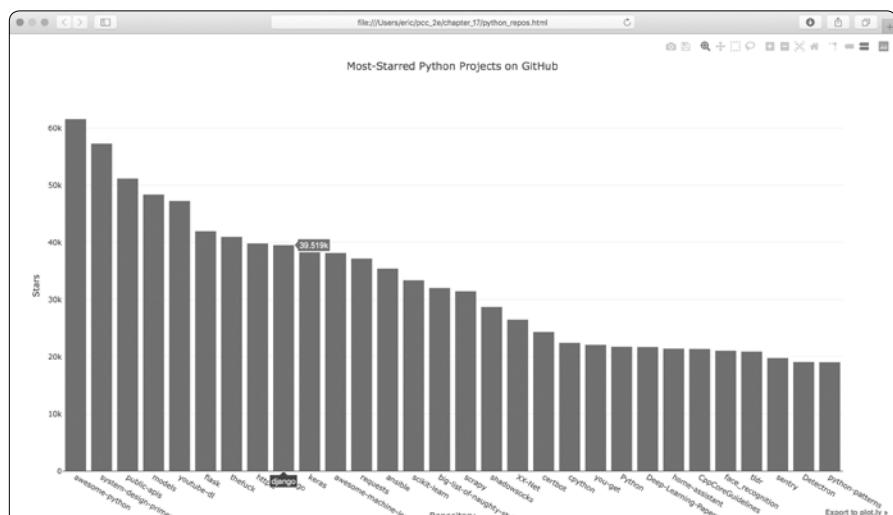


Figure 17-1: The most-starred Python projects on GitHub

Refining Plotly Charts

Let's refine the chart's styling. As you saw in Chapter 16, you can include all the styling directives as key-value pairs in the `data` and `my_layout` dictionaries.

Changes to the `data` object affect the bars. Here's a modified version of the `data` object for our chart that gives us a specific color and a clear border for each bar:

```
python_repos
_visual.py
--snip--
data = [{  
    'type': 'bar',  
    'x': repo_names,  
    'y': stars,  
    'marker': {  
        'color': 'rgb(60, 100, 150)',  
        'line': {'width': 1.5, 'color': 'rgb(25, 25, 25)'}  
    },  
    'opacity': 0.6,  
}]  
--snip--
```

The `marker` settings shown here affect the design of the bars. We set a custom blue color for the bars and specify that they'll be outlined with a dark gray line that's 1.5 pixels wide. We also set the opacity of the bars to 0.6 to soften the appearance of the chart a little.

Next, we'll modify `my_layout`:

```
python_repos
_visual.py
--snip--
my_layout = {  
    'title': 'Most-Starred Python Projects on GitHub',  
❶    'titlefont': {'size': 28},  
❷    'xaxis': {  
        'title': 'Repository',  
        'titlefont': {'size': 24},  
        'tickfont': {'size': 14},  
    },  
❸    'yaxis': {  
        'title': 'Stars',  
        'titlefont': {'size': 24},  
        'tickfont': {'size': 14},  
    },  
}  
--snip--
```

We use the `'titlefont'` key to define the font size of the overall chart title ❶. Within the `'xaxis'` dictionary, we add settings to control the font size of the x-axis title (`'titlefont'`) and also of the tick labels (`'tickfont'`) ❷. Because these are individual nested dictionaries, you can include keys for the color and font family of the axis titles and tick labels. At ❸ we define similar settings for the y-axis.

Figure 17-2 shows the restyled chart.

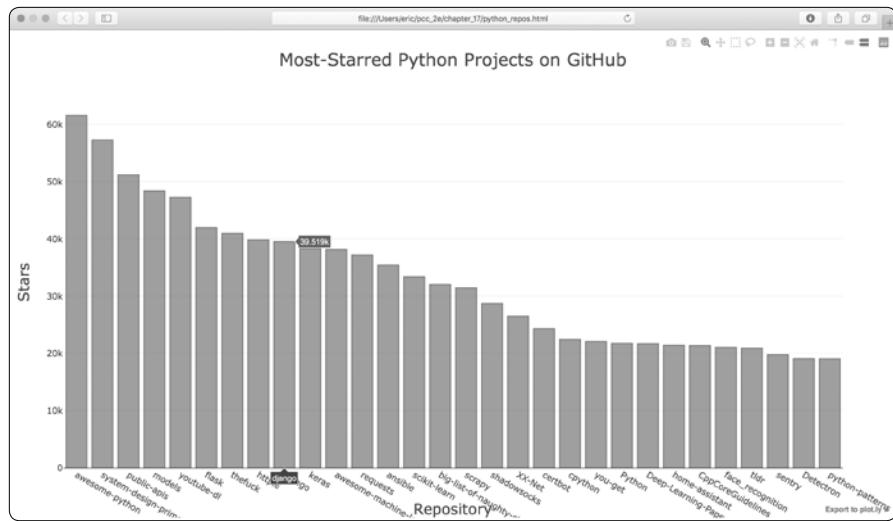


Figure 17-2: The styling for the chart has been refined.

Adding Custom Tooltips

In Plotly, you can hover the cursor over an individual bar to show the information that the bar represents. This is commonly called a *tooltip*, and in this case, it currently shows the number of stars a project has. Let's create a custom tooltip to show each project's description as well as the project's owner.

We need to pull some additional data to generate the tooltips and modify the data object:

```

python_repos
_visual.py
--snip--
# Process results.
response_dict = r.json()
repo_dicts = response_dict['items']
❶ repo_names, stars, labels = [], [], []
for repo_dict in repo_dicts:
    repo_names.append(repo_dict['name'])
    stars.append(repo_dict['stargazers_count'])

❷     owner = repo_dict['owner']['login']
    description = repo_dict['description']
❸     label = f'{owner}<br />{description}'
    labels.append(label)

# Make visualization.
data = [
    {
        'type': 'bar',
        'x': repo_names,
        'y': stars,
        '❹ 'hovertext': labels,
        'marker': {

```

```

        'color': 'rgb(60, 100, 150)',
        'line': {'width': 1.5, 'color': 'rgb(25, 25, 25)'}
    },
    'opacity': 0.6,
])
--snip--

```

We first define a new empty list, `labels`, to hold the text we want to display for each project ❶. In the loop where we process the data, we pull the owner and the description for each project ❷. Plotly allows you to use HTML code within text elements, so we generate a string for the label with a line break (`
`) between the project owner's username and the description ❸. We then store this label in the list `labels`.

In the data dictionary, we add an entry with the key '`hovertext`' and assign it the list we just created ❹. As Plotly creates each bar, it will pull labels from this list and only display them when the viewer hovers over a bar.

Figure 17-3 shows the resulting chart.

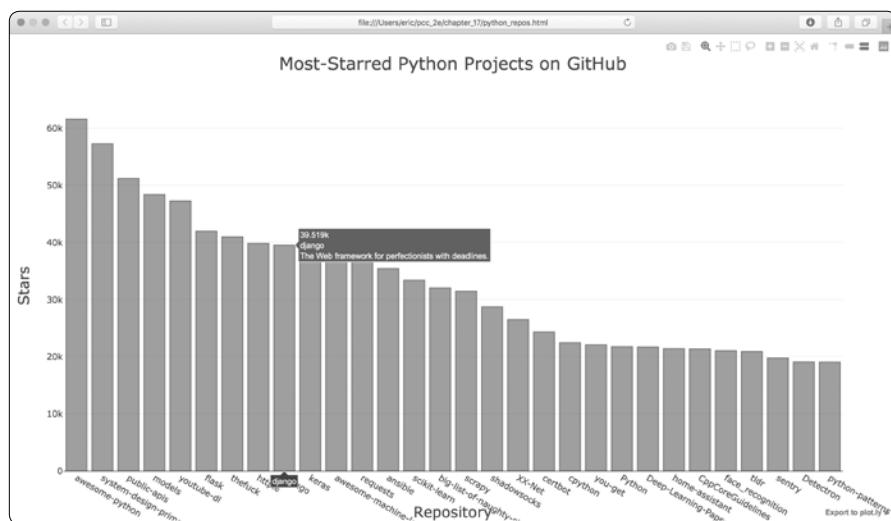


Figure 17-3: Hovering over a bar shows the project's owner and description.

Adding Clickable Links to Our Graph

Because Plotly allows you to use HTML on text elements, we can easily add links to a chart. Let's use the x-axis labels as a way to let the viewer visit any project's home page on GitHub. We need to pull the URLs from the data and use them when generating the x-axis labels:

```

python_repos
_visual.py
--snip--
# Process results.
response_dict = r.json()
repo_dicts = response_dict['items']

```

```

❶ repo_links, stars, labels = [], [], []
for repo_dict in repo_dicts:
    repo_name = repo_dict['name']
❷    repo_url = repo_dict['html_url']
❸    repo_link = f"<a href='{repo_url}'>{repo_name}</a>"
    repo_links.append(repo_link)

    stars.append(repo_dict['stargazers_count'])
--snip--

# Make visualization.
data = [{  

    'type': 'bar',  

❹    'x': repo_links,  

    'y': stars,  

    --snip--  

}]
--snip--

```

We update the name of the list we're creating from `repo_names` to `repo_links` to more accurately communicate the kind of information we're putting together for the chart ❶. We then pull the URL for the project from `repo_dict` and assign it to the temporary variable `repo_url` ❷. At ❸ we generate a link to the project. We use the HTML anchor tag, which has the form `link text`, to generate the link. We then append this link to the list `repo_links`.

At ❹ we use this list for the x-values in the chart. The result looks the same as before, but now the viewer can click any of the project names at the bottom of the chart to visit that project's home page on GitHub. Now we have an interactive, informative visualization of data retrieved through an API!

More About Plotly and the GitHub API

To read more about working with Plotly charts, there are two good places to start. You can find the *Plotly User Guide in Python* at <https://plot.ly/python/user-guide/>. This resource gives you a better understanding of how Plotly uses your data to construct a visualization and why it approaches defining data visualizations in this way.

The *python figure reference* at <https://plot.ly/python/reference/> lists all the settings you can use to configure Plotly visualizations. All the possible chart types are listed as well as all the attributes you can set for every configuration option.

For more about the GitHub API, refer to its documentation at <https://developer.github.com/v3/>. Here you'll learn how to pull a wide variety of specific information from GitHub. If you have a GitHub account, you can work with your own data as well as the publicly available data for other users' repositories.

The Hacker News API

To explore how to use API calls on other sites, let's take a quick look at Hacker News (<http://news.ycombinator.com/>). On Hacker News, people share articles about programming and technology, and engage in lively discussions about those articles. The Hacker News API provides access to data about all submissions and comments on the site, and you can use the API without having to register for a key.

The following call returns information about the current top article as of this writing:

<https://hacker-news.firebaseio.com/v0/item/19155826.json>

When you enter this URL in a browser, you'll see that the text on the page is enclosed by braces, meaning it's a dictionary. But the response is difficult to examine without some better formatting. Let's run this URL through the `json.dump()` method, like we did in the earthquake project in Chapter 16, so we can explore the kind of information that's returned about an article:

```
hn_article.py
import requests
import json

# Make an API call, and store the response.
url = 'https://hacker-news.firebaseio.com/v0/item/19155826.json'
r = requests.get(url)
print(f"Status code: {r.status_code}")

# Explore the structure of the data.
response_dict = r.json()
readable_file = 'data/readable_hn_data.json'
with open(readable_file, 'w') as f:
    json.dump(response_dict, f, indent=4)
```

Everything in this program should look familiar, because we've used it all in the previous two chapters. The output is a dictionary of information about the article with the ID 19155826:

```
readable_hn
_data.json
{
    "by": "jimktrains2",
    "descendants": 220,
    "id": 19155826,
    ② "kids": [
        19156572,
        19158857,
        --snip--
    ],
    "score": 722,
    "time": 1550085414,
    ③ "title": "Nasa's Mars Rover Opportunity Concludes a 15-Year Mission",
    "type": "story",
    ④ "url": "https://www.nytimes.com/.../mars-opportunity-rover-dead.html"
}
```

The dictionary contains a number of keys we can work with. The key 'descendants' tells us the number of comments the article has received ❶. The key 'kids' provides the IDs of all comments made directly in response to this submission ❷. Each of these comments might have comments of their own as well, so the number of descendants a submission has is usually greater than its number of kids. We can see the title of the article being discussed ❸, and a URL for the article that's being discussed as well ❹.

The following URL returns a simple list of all the IDs of the current top articles on Hacker News:

<https://hacker-news.firebaseio.com/v0/topstories.json>

We can use this call to find out which articles are on the home page right now, and then generate a series of API calls similar to the one we just examined. With this approach, we can print a summary of all the articles on the front page of Hacker News at the moment:

```
hn_submissions.py
from operator import itemgetter

import requests

# Make an API call and store the response.
❶ url = 'https://hacker-news.firebaseio.com/v0/topstories.json'
r = requests.get(url)
print(f"Status code: {r.status_code}")

# Process information about each submission.
❷ submission_ids = r.json()
❸ submission_dicts = []
for submission_id in submission_ids[:30]:
    # Make a separate API call for each submission.
    ❹ url = f"https://hacker-news.firebaseio.com/v0/item/{submission_id}.json"
    r = requests.get(url)
    print(f"id: {submission_id}\tstatus: {r.status_code}")
    response_dict = r.json()

    # Build a dictionary for each article.
    ❺ submission_dict = {
        'title': response_dict['title'],
        'hn_link': f"http://news.ycombinator.com/item?id={submission_id}",
        'comments': response_dict['descendants'],
    }
    ❻ submission_dicts.append(submission_dict)

❼ submission_dicts = sorted(submission_dicts, key=itemgetter('comments'),
                             reverse=True)

➋ for submission_dict in submission_dicts:
    print(f"\nTitle: {submission_dict['title']}")
    print(f"Discussion link: {submission_dict['hn_link']}")
    print(f"Comments: {submission_dict['comments']}")
```

First, we make an API call, and then print the status of the response ❶. This API call returns a list containing the IDs of up to the 500 most popular articles on Hacker News at the time the call is issued. We then convert the response object to a Python list at ❷, which we store in `submission_ids`. We'll use these IDs to build a set of dictionaries that each store information about one of the current submissions.

We set up an empty list called `submission_dicts` at ❸ to store these dictionaries. We then loop through the IDs of the top 30 submissions. We make a new API call for each submission by generating a URL that includes the current value of `submission_id` ❹. We print the status of each request along with its ID, so we can see whether it's successful.

At ❽ we create a dictionary for the submission currently being processed, where we store the title of the submission, a link to the discussion page for that item, and the number of comments the article has received so far. Then we append each `submission_dict` to the list `submission_dicts` ❾.

Each submission on Hacker News is ranked according to an overall score based on a number of factors including how many times it's been voted up, how many comments it's received, and how recent the submission is. We want to sort the list of dictionaries by the number of comments. To do this, we use a function called `itemgetter()` ❿, which comes from the `operator` module. We pass this function the key '`'comments'`', and it pulls the value associated with that key from each dictionary in the list. The `sorted()` function then uses this value as its basis for sorting the list. We sort the list in reverse order to place the most-commented stories first.

Once the list is sorted, we loop through the list at ❻ and print out three pieces of information about each of the top submissions: the title, a link to the discussion page, and the number of comments the submission currently has:

```
Status code: 200
id: 19155826    status: 200
id: 19180181    status: 200
id: 19181473    status: 200
--snip--

Title: Nasa's Mars Rover Opportunity Concludes a 15-Year Mission
Discussion link: http://news.ycombinator.com/item?id=19155826
Comments: 220

Title: Ask HN: Is it practical to create a software-controlled model rocket?
Discussion link: http://news.ycombinator.com/item?id=19180181
Comments: 72

Title: Making My Own USB Keyboard from Scratch
Discussion link: http://news.ycombinator.com/item?id=19181473
Comments: 62
--snip--
```

You would use a similar process to access and analyze information with any API. With this data, you could make a visualization showing which submissions have inspired the most active recent discussions. This is also the basis for apps that provide a customized reading experience for sites like Hacker News. To learn more about what kind of information you can access through the Hacker News API, visit the documentation page at <https://github.com/HackerNews/API/>.

TRY IT YOURSELF

17-1. Other Languages: Modify the API call in `python_repos.py` so it generates a chart showing the most popular projects in other languages. Try languages such as *JavaScript*, *Ruby*, *C*, *Java*, *Perl*, *Haskell*, and *Go*.

17-2. Active Discussions: Using the data from `hn_submissions.py`, make a bar chart showing the most active discussions currently happening on Hacker News. The height of each bar should correspond to the number of comments each submission has. The label for each bar should include the submission's title and should act as a link to the discussion page for that submission.

17-3. Testing `python_repos.py`: In `python_repos.py`, we printed the value of `status_code` to make sure the API call was successful. Write a program called `test_python_repos.py` that uses `unittest` to assert that the value of `status_code` is 200. Figure out some other assertions you can make—for example, that the number of items returned is expected and that the total number of repositories is greater than a certain amount.

17-4. Further Exploration: Visit the documentation for Plotly and either the GitHub API or the Hacker News API. Use some of the information you find there to either customize the style of the plots we've already made or pull some different information and create your own visualizations.

Summary

In this chapter, you learned how to use APIs to write self-contained programs that automatically gather the data they need and use that data to create a visualization. You used the GitHub API to explore the most-starred Python projects on GitHub, and you also looked briefly at the Hacker News API. You learned how to use the Requests package to automatically issue an API call to GitHub and how to process the results of that call. Some Plotly settings were also introduced that further customize the appearance of the charts you generate.

In the next chapter, you'll use Django to build a web application as your final project.

PROJECT 3

WEB APPLICATIONS

18

GETTING STARTED WITH DJANGO



Behind the scenes, today's websites are rich applications that act like fully developed desktop applications. Python has a great set of tools called Django for building web applications. Django is a *web framework*—a set of tools designed to help you build interactive websites. In this chapter, you'll learn how to use Django ([https://djangoproject.com/](https://.djangoproject.com/)) to build a project called Learning Log—an online journal system that lets you keep track of information you've learned about particular topics.

We'll write a specification for this project, and then we'll define models for the data the app will work with. We'll use Django's admin system to enter some initial data, and then you'll learn to write views and templates so Django can build the site's pages.

Django can respond to page requests and make it easier to read and write to a database, manage users, and much more. In Chapters 19 and 20, you'll refine the Learning Log project and then deploy it to a live server so you (and your friends) can use it.

Setting Up a Project

When beginning a project, you first need to describe the project in a specification, or *spec*. Then you'll set up a virtual environment in which to build the project.

Writing a Spec

A full spec details the project goals, describes the project's functionality, and discusses its appearance and user interface. Like any good project or business plan, a spec should keep you focused and help keep your project on track. We won't write a full project spec here, but we'll lay out a few clear goals to keep the development process focused. Here's the spec we'll use:

We'll write a web app called Learning Log that allows users to log the topics they're interested in and to make journal entries as they learn about each topic. The Learning Log home page will describe the site and invite users to either register or log in. Once logged in, a user can create new topics, add new entries, and read and edit existing entries.

When you learn about a new topic, keeping a journal of what you've learned can be helpful in tracking and revisiting information. A good app makes this process efficient.

Creating a Virtual Environment

To work with Django, we'll first set up a virtual environment. A *virtual environment* is a place on your system where you can install packages and isolate them from all other Python packages. Separating one project's libraries from other projects is beneficial and will be necessary when we deploy Learning Log to a server in Chapter 20.

Create a new directory for your project called *learning_log*, switch to that directory in a terminal, and enter the following code to create a virtual environment:

```
learning_log$ python -m venv ll_env
learning_log$
```

Here we're running the `venv` virtual environment module and using it to create a virtual environment named *ll_env* (note that this is *ll_env* with two lowercase *Ls*, not two ones). If you use a command such as `python3` when running programs or installing packages, make sure to use that command here.

Activating the Virtual Environment

Now we need to activate the virtual environment using the following command:

```
learning_log$ source ll_env/bin/activate  
❶ (ll_env)learning_log$
```

This command runs the script `activate` in `ll_env/bin`. When the environment is active, you'll see the name of the environment in parentheses, as shown at ❶; then you can install packages to the environment and use packages that have already been installed. Packages you install in `ll_env` will be available only while the environment is active.

NOTE *If you're using Windows, use the command `ll_env\Scripts\activate` (without the word `source`) to activate the virtual environment. If you're using PowerShell, you might need to capitalize `Activate`.*

To stop using a virtual environment, enter `deactivate`:

```
(ll_env)learning_log$ deactivate  
learning_log$
```

The environment will also become inactive when you close the terminal it's running in.

Installing Django

Once the virtual environment is activated, enter the following to install Django:

```
(ll_env)learning_log$ pip install django  
Collecting django  
--snip--  
Installing collected packages: pytz, django  
Successfully installed django-2.2.0 pytz-2018.9 sqlparse-0.2.4  
(ll_env)learning_log$
```

Because we're working in a virtual environment, which is its own self-contained environment, this command is the same on all systems. There's no need to use the `--user` flag, and there's no need to use longer commands, such as `python -m pip install package_name`.

Keep in mind that Django will be available only when the `ll_env` environment is active.

NOTE *Django releases a new version about every eight months, so you may see a newer version when you install Django. This project will most likely work as it's written here, even on newer versions of Django. If you want to make sure to use the same version of Django you see here, use the command `pip install django==2.2.*`. This will install the latest release of Django 2.2. If you have any issues related to the version you're using, see the online resources for the book at <https://nostarch.com/pythoncrashcourse2e/>.*

Creating a Project in Django

Without leaving the active virtual environment (remember to look for `ll_env` in parentheses in the terminal prompt), enter the following commands to create a new project:

```
❶ (ll_env)learning_log$ django-admin startproject learning_log .
❷ (ll_env)learning_log$ ls
    learning_log ll_env manage.py
❸ (ll_env)learning_log$ ls learning_log
    __init__.py settings.py urls.py wsgi.py
```

The command at ❶ tells Django to set up a new project called `learning_log`. The dot at the end of the command creates the new project with a directory structure that will make it easy to deploy the app to a server when we're finished developing it.

NOTE

Don't forget this dot, or you might run into some configuration issues when you deploy the app. If you forget the dot, delete the files and folders that were created (except ll_env), and run the command again.

Running the `ls` command (dir on Windows) ❷ shows that Django has created a new directory called `learning_log`. It also created a `manage.py` file, which is a short program that takes in commands and feeds them to the relevant part of Django to run them. We'll use these commands to manage tasks, such as working with databases and running servers.

The `learning_log` directory contains four files ❸; the most important are `settings.py`, `urls.py`, and `wsgi.py`. The `settings.py` file controls how Django interacts with your system and manages your project. We'll modify a few of these settings and add some settings of our own as the project evolves. The `urls.py` file tells Django which pages to build in response to browser requests. The `wsgi.py` file helps Django serve the files it creates. The filename is an acronym for *web server gateway interface*.

Creating the Database

Django stores most of the information for a project in a database, so next we need to create a database that Django can work with. Enter the following command (still in an active environment):

```
(ll_env)learning_log$ python manage.py migrate
❶ Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
  Running migrations:
    Applying contenttypes.0001_initial... OK
    Applying auth.0001_initial... OK
    --snip--
    Applying sessions.0001_initial... OK
❷ (ll_env)learning_log$ ls
    db.sqlite3  learning_log  ll_env  manage.py
```

Any time we modify a database, we say we're *migrating* the database. Issuing the `migrate` command for the first time tells Django to make sure the database matches the current state of the project. The first time we run this command in a new project using SQLite (more about SQLite in a moment), Django will create a new database for us. At ❶, Django reports that it will prepare the database to store information it needs to handle administrative and authentication tasks.

Running the `ls` command shows that Django created another file called `db.sqlite3` ❷. SQLite is a database that runs off a single file; it's ideal for writing simple apps because you won't have to pay much attention to managing the database.

NOTE

In an active virtual environment, use the command `python` to run `manage.py` commands, even if you use something different, like `python3`, to run other programs. In a virtual environment, the command `python` refers to the version of Python that created the virtual environment.

Viewing the Project

Let's make sure that Django has set up the project properly. Enter the `runserver` command as follows to view the project in its current state:

```
(11_env)learning_log$ python manage.py runserver
Watchman unavailable: pywatchman not installed.
Watching for file changes with StatReloader
Performing system checks...
```

- ❶ System check identified no issues (0 silenced).
February 18, 2019 - 16:26:07
 - ❷ Django version 2.2.0, using settings 'learning_log.settings'
 - ❸ Starting development server at <http://127.0.0.1:8000/>
Quit the server with CONTROL-C.
-

Django should start a server called the *development server*, so you can view the project on your system to see how well it works. When you request a page by entering a URL in a browser, the Django server responds to that request by building the appropriate page and sending it to the browser.

At ❶, Django checks to make sure the project is set up properly; at ❷ it reports the version of Django in use and the name of the settings file in use; and at ❸ it reports the URL where the project is being served. The URL `http://127.0.0.1:8000/` indicates that the project is listening for requests on port 8000 on your computer, which is called a localhost. The term *localhost* refers to a server that only processes requests on your system; it doesn't allow anyone else to see the pages you're developing.

Open a web browser and enter the URL `http://localhost:8000/`, or `http://127.0.0.1:8000/` if the first one doesn't work. You should see something like Figure 18-1, a page that Django creates to let you know all is working properly so far. Keep the server running for now, but when you want to stop the server, press CTRL-C in the terminal where the `runserver` command was issued.

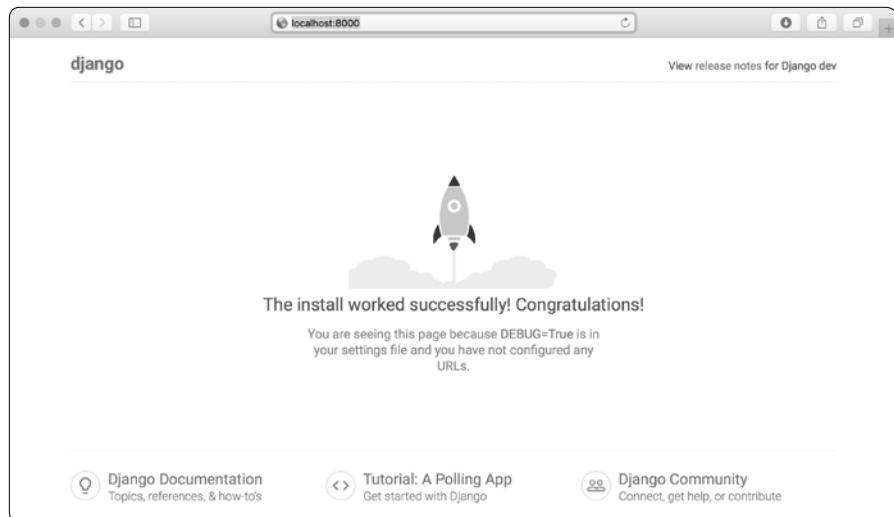


Figure 18-1: Everything is working so far.

NOTE

If you receive the error message That port is already in use, tell Django to use a different port by entering `python manage.py runserver 8001`, and then cycle through higher numbers until you find an open port.

TRY IT YOURSELF

18-1. New Projects: To get a better idea of what Django does, build a couple of empty projects and look at what Django creates. Make a new folder with a simple name, like `snap_gram` or `insta_chat` (outside of your `learning_log` directory), navigate to that folder in a terminal, and create a virtual environment. Install Django and run the command `django-admin.py startproject snap_gram` . (make sure you include the dot at the end of the command).

Look at the files and folders this command creates, and compare them to Learning Log. Do this a few times until you're familiar with what Django creates when starting a new project. Then delete the project directories if you wish.

Starting an App

A Django *project* is organized as a group of individual *apps* that work together to make the project work as a whole. For now, we'll create just one app to do most of our project's work. We'll add another app in Chapter 19 to manage user accounts.

You should leave the development server running in the terminal window you opened earlier. Open a new terminal window (or tab), and navigate to the directory that contains `manage.py`. Activate the virtual environment, and then run the `startapp` command:

```
learning_log$ source ll_env/bin/activate
(ll_env)learning_log$ python manage.py startapp learning_logs
❶ (ll_env)learning_log$ ls
db.sqlite3  learning_log  learning_logs  ll_env  manage.py
❷ (ll_env)learning_log$ ls learning_logs/
__init__.py  admin.py  apps.py  migrations  models.py  tests.py  views.py
```

The command `startapp appname` tells Django to create the infrastructure needed to build an app. When you look in the project directory now, you'll see a new folder called `learning_logs` ❶. Open that folder to see what Django has created ❷. The most important files are `models.py`, `admin.py`, and `views.py`. We'll use `models.py` to define the data we want to manage in our app. We'll look at `admin.py` and `views.py` a little later.

Defining Models

Let's think about our data for a moment. Each user will need to create a number of topics in their learning log. Each entry they make will be tied to a topic, and these entries will be displayed as text. We'll also need to store the timestamp of each entry, so we can show users when they made each entry.

Open the file `models.py`, and look at its existing content:

```
models.py
from django.db import models

# Create your models here.
```

A module called `models` is being imported for us, and we're being invited to create models of our own. A `model` tells Django how to work with the data that will be stored in the app. Code-wise, a model is just a class; it has attributes and methods, just like every class we've discussed. Here's the model for the topics users will store:

```
from django.db import models

class Topic(models.Model):
    """A topic the user is learning about."""
❶    text = models.CharField(max_length=200)
❷    date_added = models.DateTimeField(auto_now_add=True)

❸    def __str__(self):
        """Return a string representation of the model."""
        return self.text
```

We've created a class called `Topic`, which inherits from `Model`—a parent class included in Django that defines a model's basic functionality. We add two attributes to the `Topic` class: `text` and `date_added`.

The `text` attribute is a `CharField`—a piece of data that's made up of characters, or `text` ❶. You use `CharField` when you want to store a small amount of text, such as a name, a title, or a city. When we define a `CharField` attribute, we have to tell Django how much space it should reserve in the database. Here we give it a `max_length` of 200 characters, which should be enough to hold most topic names.

The `date_added` attribute is a `DateTimeField`—a piece of data that will record a date and time ❷. We pass the argument `auto_now_add=True`, which tells Django to automatically set this attribute to the current date and time whenever the user creates a new topic.

NOTE

To see the different kinds of fields you can use in a model, see the Django Model Field Reference at <https://docs.djangoproject.com/en/2.2/ref/models/fields/>. You won't need all the information right now, but it will be extremely useful when you're developing your own apps.

We tell Django which attribute to use by default when it displays information about a topic. Django calls a `_str_()` method to display a simple representation of a model. Here we've written a `_str_()` method that returns the string stored in the `text` attribute ❸.

Activating Models

To use our models, we have to tell Django to include our app in the overall project. Open `settings.py` (in the `learning_log/learning_log` directory); you'll see a section that tells Django which apps are installed and work together in the project:

```
settings.py
--snip--
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
--snip--
```

Add our app to this list by modifying `INSTALLED_APPS` so it looks like this:

```
--snip--
INSTALLED_APPS = [
    # My apps
    'learning_logs',

    # Default django apps.
    'django.contrib.admin',
```

```
--snip--  
]  
--snip--
```

Grouping apps together in a project helps to keep track of them as the project grows to include more apps. Here we start a section called *My apps*, which includes only `learning_logs` for now. It's important to place your own apps before the default apps in case you need to override any behavior of the default apps with your own custom behavior.

Next, we need to tell Django to modify the database so it can store information related to the model `Topic`. From the terminal, run the following command:

```
(11_env)learning_log$ python manage.py makemigrations learning_logs  
Migrations for 'learning_logs':  
    learning_logs/migrations/0001_initial.py  
        - Create model Topic  
(11_env)learning_log$
```

The command `makemigrations` tells Django to figure out how to modify the database so it can store the data associated with any new models we've defined. The output here shows that Django has created a migration file called `0001_initial.py`. This migration will create a table for the model `Topic` in the database.

Now we'll apply this migration and have Django modify the database for us:

```
(11_env)learning_log$ python manage.py migrate  
Operations to perform:  
    Apply all migrations: admin, auth, contenttypes, learning_logs, sessions  
Running migrations:  
① Applying learning_logs.0001_initial... OK
```

Most of the output from this command is identical to the first time we issued the `migrate` command. The line we need to check appears at ①, where Django confirms that the migration for `learning_logs` worked `OK`.

Whenever we want to modify the data that Learning Log manages, we'll follow these three steps: modify `models.py`, call `makemigrations` on `learning_logs`, and tell Django to `migrate` the project.

The Django Admin Site

Django makes it easy to work with your models through the *admin site*. Only the site's administrators use the admin site, not general users. In this section, we'll set up the admin site and use it to add some topics through the `Topic` model.

Setting Up a Superuser

Django allows you to create a *superuser*, a user who has all privileges available on the site. A user's *privileges* control the actions that user can take.

The most restrictive privilege settings allow a user to only read public information on the site. Registered users typically have the privilege of reading their own private data and some selected information available only to members. To effectively administer a web application, the site owner usually needs access to all information stored on the site. A good administrator is careful with their users' sensitive information, because users put a lot of trust into the apps they access.

To create a superuser in Django, enter the following command and respond to the prompts:

```
(ll_env)learning_log$ python manage.py createsuperuser
❶ Username (leave blank to use 'eric'): ll_admin
❷ Email address:
❸ Password:
Password (again):
Superuser created successfully.
(ll_env)learning_log$
```

When you issue the command `createsuperuser`, Django prompts you to enter a username for the superuser ❶. Here I'm using `ll_admin`, but you can enter any username you want. You can enter an email address if you want or just leave this field blank ❷. You'll need to enter your password twice ❸.

NOTE

Some sensitive information can be hidden from a site's administrators. For example, Django doesn't store the password you enter; instead, it stores a string derived from the password, called a hash. Each time you enter your password, Django hashes your entry and compares it to the stored hash. If the two hashes match, you're authenticated. By requiring hashes to match, if an attacker gains access to a site's database, they'll be able to read its stored hashes but not the passwords. When a site is set up properly, it's almost impossible to get the original passwords from the hashes.

Registering a Model with the Admin Site

Django includes some models in the admin site automatically, such as `User` and `Group`, but the models we create need to be added manually.

When we started the `learning_logs` app, Django created an `admin.py` file in the same directory as `models.py`. Open the `admin.py` file:

```
admin.py  from django.contrib import admin
          # Register your models here.
```

To register `Topic` with the admin site, enter the following:

```
from django.contrib import admin
❶ from .models import Topic
❷ admin.site.register(Topic)
```

This code first imports the model we want to register, `Topic` ❶. The dot in front of `models` tells Django to look for `models.py` in the same directory as `admin.py`. The code `admin.site.register()` tells Django to manage our model through the admin site ❷.

Now use the superuser account to access the admin site. Go to `http://localhost:8000/admin/`, and enter the username and password for the superuser you just created. You should see a screen like the one in Figure 18-2. This page allows you to add new users and groups, and change existing ones. You can also work with data related to the `Topic` model that we just defined.

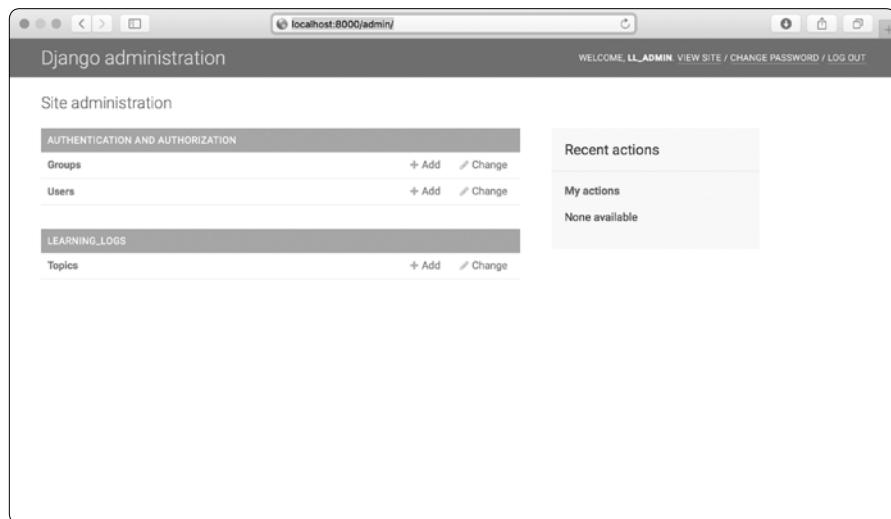


Figure 18-2: The admin site with Topic included

NOTE

If you see a message in your browser that the web page is not available, make sure you still have the Django server running in a terminal window. If you don't, activate a virtual environment and reissue the command `python manage.py runserver`. If you're having trouble viewing your project at any point in the development process, closing any open terminals and reissuing the `runserver` command is a good first troubleshooting step.

Adding Topics

Now that `Topic` has been registered with the admin site, let's add our first topic. Click **Topics** to go to the Topics page, which is mostly empty, because we have no topics to manage yet. Click **Add Topic**, and a form for adding a new topic appears. Enter **Chess** in the first box and click **Save**. You'll be sent back to the Topics admin page, and you'll see the topic you just created.

Let's create a second topic so we'll have more data to work with. Click **Add Topic** again, and enter **Rock Climbing**. Click **Save**, and you'll be sent back to the main Topics page again. Now you'll see Chess and Rock Climbing listed.

Defining the Entry Model

For a user to record what they've been learning about chess and rock climbing, we need to define a model for the kinds of entries users can make in their learning logs. Each entry needs to be associated with a particular topic. This relationship is called a *many-to-one relationship*, meaning many entries can be associated with one topic.

Here's the code for the `Entry` model. Place it in your `models.py` file:

`models.py`

```
from django.db import models

class Topic(models.Model):
    --snip--

❶ class Entry(models.Model):
    """Something specific learned about a topic."""
❷     topic = models.ForeignKey(Topic, on_delete=models.CASCADE)
❸     text = models.TextField()
        date_added = models.DateTimeField(auto_now_add=True)

❹     class Meta:
        verbose_name_plural = 'entries'

❺     def __str__(self):
        """Return a string representation of the model."""
        return f'{self.text[:50]}...'
```

The `Entry` class inherits from Django's base `Model` class, just as `Topic` did ❶. The first attribute, `topic`, is a `ForeignKey` instance ❷. A *foreign key* is a database term; it's a reference to another record in the database. This is the code that connects each entry to a specific topic. Each topic is assigned a key, or ID, when it's created. When Django needs to establish a connection between two pieces of data, it uses the key associated with each piece of information. We'll use these connections shortly to retrieve all the entries associated with a certain topic. The `on_delete=models.CASCADE` argument tells Django that when a topic is deleted, all the entries associated with that topic should be deleted as well. This is known as a *cascading delete*.

Next is an attribute called `text`, which is an instance of `TextField` ❸. This kind of field doesn't need a size limit, because we don't want to limit the size of individual entries. The `date_added` attribute allows us to present entries in the order they were created and to place a timestamp next to each entry.

At ❹ we nest the `Meta` class inside our `Entry` class. The `Meta` class holds extra information for managing a model; here, it allows us to set a special attribute telling Django to use `Entries` when it needs to refer to more than one entry. Without this, Django would refer to multiple entries as `Entry`s.

The `__str__()` method tells Django which information to show when it refers to individual entries. Because an entry can be a long body of text, we tell Django to show just the first 50 characters of `text` ❺. We also add an ellipsis to clarify that we're not always displaying the entire entry.

Migrating the Entry Model

Because we've added a new model, we need to migrate the database again. This process will become quite familiar: you modify `models.py`, run the command `python manage.py makemigrations app_name`, and then run the command `python manage.py migrate`.

Migrate the database and check the output by entering the following commands:

```
(11_env)learning_log$ python manage.py makemigrations learning_logs
Migrations for 'learning_logs':
❶  learning_logs/migrations/0002_entry.py
    - Create model Entry
(11_env)learning_log$ python manage.py migrate
Operations to perform:
--snip--
❷  Applying learning_logs.0002_entry... OK
```

A new migration called `0002_entry.py` is generated, which tells Django how to modify the database to store information related to the model `Entry` ❶. When we issue the `migrate` command, we see that Django applied this migration, and everything was okay ❷.

Registering Entry with the Admin Site

We also need to register the `Entry` model. Here's what `admin.py` should look like now:

```
admin.py
from django.contrib import admin
from .models import Topic, Entry
admin.site.register(Topic)
admin.site.register(Entry)
```

Go back to `http://localhost/admin/`, and you should see `Entries` listed under `Learning_Logs`. Click the **Add** link for Entries, or click **Entries**, and then choose **Add entry**. You should see a drop-down list to select the topic you're creating an entry for and a text box for adding an entry. Select **Chess** from the drop-down list, and add an entry. Here's the first entry I made:

The opening is the first part of the game, roughly the first ten moves or so. In the opening, it's a good idea to do three things—bring out your bishops and knights, try to control the center of the board, and castle your king.

Of course, these are just guidelines. It will be important to learn when to follow these guidelines and when to disregard these suggestions.

When you click **Save**, you'll be brought back to the main admin page for entries. Here, you'll see the benefit of using `text[:50]` as the string representation for each entry; it's much easier to work with multiple entries in the admin interface if you see only the first part of an entry rather than the entire text of each entry.

Make a second entry for Chess and one entry for Rock Climbing so we have some initial data. Here's a second entry for Chess:

In the opening phase of the game, it's important to bring out your bishops and knights. These pieces are powerful and maneuverable enough to play a significant role in the beginning moves of a game.

And here's a first entry for Rock Climbing:

One of the most important concepts in climbing is to keep your weight on your feet as much as possible. There's a myth that climbers can hang all day on their arms. In reality, good climbers have practiced specific ways of keeping their weight over their feet whenever possible.

These three entries will give us something to work with as we continue to develop Learning Log.

The Django Shell

With some data entered, we can examine that data programmatically through an interactive terminal session. This interactive environment is called the *Django shell*, and it's a great environment for testing and troubleshooting your project. Here's an example of an interactive shell session:

```
(11_env)learning_log$ python manage.py shell
❶ >>> from learning_logs.models import Topic
>>> Topic.objects.all()
<QuerySet [<Topic: Chess>, <Topic: Rock Climbing>]>
```

The command `python manage.py shell`, run in an active virtual environment, launches a Python interpreter that you can use to explore the data stored in your project's database. Here, we import the model `Topic` from the `learning_logs.models` module ❶. We then use the method `Topic.objects.all()` to get all the instances of the model `Topic`; the list that's returned is called a *queryset*.

We can loop over a queryset just as we'd loop over a list. Here's how you can see the ID that's been assigned to each topic object:

```
>>> topics = Topic.objects.all()
>>> for topic in topics:
...     print(topic.id, topic)
```

```
...
1 Chess
2 Rock Climbing
```

We store the queryset in `topics`, and then print each topic's `id` attribute and the string representation of each topic. We can see that Chess has an ID of 1, and Rock Climbing has an ID of 2.

If you know the ID of a particular object, you can use the method `Topic.objects.get()` to retrieve that object and examine any attribute the object has. Let's look at the text and `date_added` values for Chess:

```
>>> t = Topic.objects.get(id=1)
>>> t.text
'Chess'
>>> t.date_added
datetime.datetime(2019, 2, 19, 1, 55, 31, 98500, tzinfo=<UTC>)
```

We can also look at the entries related to a certain topic. Earlier we defined the `topic` attribute for the `Entry` model. This was a `ForeignKey`, a connection between each entry and a topic. Django can use this connection to get every entry related to a certain topic, like this:

```
❶ >>> t.entry_set.all()
<QuerySet [The opening is the first part of the game, roughly...>,
<Entry:
In the opening phase of the game, it's important t....>]>
```

To get data through a foreign key relationship, you use the lowercase name of the related model followed by an underscore and the word `set` ❶. For example, say you have the models `Pizza` and `Topping`, and `Topping` is related to `Pizza` through a foreign key. If your object is called `my_pizza`, representing a single pizza, you can get all of the pizza's toppings using the code `my_pizza.topping_set.all()`.

We'll use this kind of syntax when we begin to code the pages users can request. The shell is very useful for making sure your code retrieves the data you want it to. If your code works as you expect it to in the shell, you can expect it to work properly in the files within your project. If your code generates errors or doesn't retrieve the data you expect it to, it's much easier to troubleshoot your code in the simple shell environment than within the files that generate web pages. We won't refer to the shell much, but you should continue using it to practice working with Django's syntax for accessing the data stored in the project.

NOTE

Each time you modify your models, you'll need to restart the shell to see the effects of those changes. To exit a shell session, press `CTRL-D`; on Windows, press `CTRL-Z` and then press `ENTER`.

TRY IT YOURSELF

18-2. Short Entries: The `__str__()` method in the `Entry` model currently appends an ellipsis to every instance of `Entry` when Django shows it in the admin site or the shell. Add an `if` statement to the `__str__()` method that adds an ellipsis only if the entry is longer than 50 characters. Use the admin site to add an entry that's fewer than 50 characters in length, and check that it doesn't have an ellipsis when viewed.

18-3. The Django API: When you write code to access the data in your project, you're writing a *query*. Skim through the documentation for querying your data at <https://docs.djangoproject.com/en/2.2/topics/db/queries/>. Much of what you see will look new to you, but it will be very useful as you start to work on your own projects.

18-4. Pizzeria: Start a new project called `pizzeria` with an app called `pizzas`. Define a model `Pizza` with a field called `name`, which will hold name values, such as `Hawaiian` and `Meat Lovers`. Define a model called `Topping` with fields `name` and `pizza`. The `pizza` field should be a foreign key to `Pizza`, and `name` should be able to hold values such as `pineapple`, `Canadian bacon`, and `sausage`.

Register both models with the admin site, and use the site to enter some pizza names and toppings. Use the shell to explore the data you entered.

Making Pages: The Learning Log Home Page

Making web pages with Django consists of three stages: defining URLs, writing views, and writing templates. You can do these in any order, but in this project we'll always start by defining the URL pattern. A URL pattern describes the way the URL is laid out. It also tells Django what to look for when matching a browser request with a site URL so it knows which page to return.

Each URL then maps to a particular *view*—the view function retrieves and processes the data needed for that page. The view function often renders the page using a *template*, which contains the overall structure of the page. To see how this works, let's make the home page for Learning Log. We'll define the URL for the home page, write its view function, and create a simple template.

Because all we're doing is making sure Learning Log works as it's supposed to, we'll make a simple page for now. A functioning web app is fun to style when it's complete; an app that looks good but doesn't work well is pointless. For now, the home page will display only a title and a brief description.

Mapping a URL

Users request pages by entering URLs into a browser and clicking links, so we'll need to decide what URLs are needed. The home page URL is first: it's the base URL people use to access the project. At the moment the base URL, `http://localhost:8000/`, returns the default Django site that lets us know the project was set up correctly. We'll change this by mapping the base URL to Learning Log's home page.

In the main `learning_log` project folder, open the file `urls.py`. Here's the code you should see:

```
urls.py ❶ from django.contrib import admin
        from django.urls import path

❷ urlpatterns = [
❸     path('admin/', admin.site.urls),
]
```

The first two lines import a module and a function to manage URLs for the admin site ❶. The body of the file defines the `urlpatterns` variable ❷. In this `urls.py` file, which represents the project as a whole, the `urlpatterns` variable includes sets of URLs from the apps in the project. The code at ❸ includes the module `admin.site.urls`, which defines all the URLs that can be requested from the admin site.

We need to include the URLs for `learning_logs`, so add the following:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
❶    path('', include('learning_logs.urls')),
]
```

We've added a line to include the module `learning_logs.urls` at ❶.

The default `urls.py` is in the `learning_log` folder; now we need to make a second `urls.py` file in the `learning_logs` folder. Create a new Python file and save it as `urls.py` in `learning_logs`, and enter this code into it:

```
urls.py ❶ """Defines URL patterns for learning_logs."""
❷ from django.urls import path
❸ from . import views

❹ app_name = 'learning_logs'
❺ urlpatterns = [
    # Home page
❻    path('', views.index, name='index'),
]
```

To make it clear which *urls.py* we’re working in, we add a docstring at the beginning of the file ❶. We then import the `path` function, which is needed when mapping URLs to views ❷. We also import the `views` module ❸; the dot tells Python to import the *views.py* module from the same directory as the current *urls.py* module. The variable `app_name` helps Django distinguish this *urls.py* file from files of the same name in other apps within the project ❹. The variable `urlpatterns` in this module is a list of individual pages that can be requested from the *learning_logs* app ❺.

The actual URL pattern is a call to the `path()` function, which takes three arguments ❻. The first argument is a string that helps Django route the current request properly. Django receives the requested URL and tries to route the request to a view. It does this by searching all the URL patterns we’ve defined to find one that matches the current request. Django ignores the base URL for the project (`http://localhost:8000/`), so the empty string ('') matches the base URL. Any other URL won’t match this pattern, and Django will return an error page if the URL requested doesn’t match any existing URL patterns.

The second argument in `path()` ❼ specifies which function to call in *views.py*. When a requested URL matches the pattern we’re defining, Django calls the `index()` function from *views.py* (we’ll write this view function in the next section). The third argument provides the name `index` for this URL pattern so we can refer to it in other code sections. Whenever we want to provide a link to the home page, we’ll use this name instead of writing out a URL.

Writing a View

A view function takes in information from a request, prepares the data needed to generate a page, and then sends the data back to the browser, often by using a template that defines what the page will look like.

The file *views.py* in *learning_logs* was generated automatically when we ran the command `python manage.py startapp`. Here’s what’s in *views.py* right now:

```
views.py      from django.shortcuts import render

# Create your views here.
```

Currently, this file just imports the `render()` function, which renders the response based on the data provided by views. Open the views file and add the following code for the home page:

```
from django.shortcuts import render

def index(request):
    """The home page for Learning Log."""
    return render(request, 'learning_logs/index.html')
```

When a URL request matches the pattern we just defined, Django looks for a function called `index()` in the *views.py* file. Django then passes the

request object to this view function. In this case, we don't need to process any data for the page, so the only code in the function is a call to render(). The render() function here passes two arguments—the original request object and a template it can use to build the page. Let's write this template.

Writing a Template

The template defines what the page should look like, and Django fills in the relevant data each time the page is requested. A template allows you to access any data provided by the view. Because our view for the home page provided no data, this template is fairly simple.

Inside the `learning_logs` folder, make a new folder called `templates`. Inside the `templates` folder, make another folder called `learning_logs`. This might seem a little redundant (we have a folder named `learning_logs` inside a folder named `templates` inside a folder named `learning_logs`), but it sets up a structure that Django can interpret unambiguously, even in the context of a large project containing many individual apps. Inside the inner `learning_logs` folder, make a new file called `index.html`. The path to the file will be `learning_log/learning_logs/templates/learning_logs/index.html`. Enter the following code into that file:

`index.html`

```
<p>Learning Log</p>

<p>Learning Log helps you keep track of your learning, for any topic you're
learning about.</p>
```

This is a very simple file. If you're not familiar with HTML, the `<p></p>` tags signify paragraphs. The `<p>` tag opens a paragraph, and the `</p>` tag closes a paragraph. We have two paragraphs: the first acts as a title, and the second describes what users can do with Learning Log.

Now when you request the project's base URL, `http://localhost:8000/`, you should see the page we just built instead of the default Django page. Django will take the requested URL, and that URL will match the pattern ''; then Django will call the function `views.index()`, which will render the page using the template contained in `index.html`. Figure 18-3 shows the resulting page.

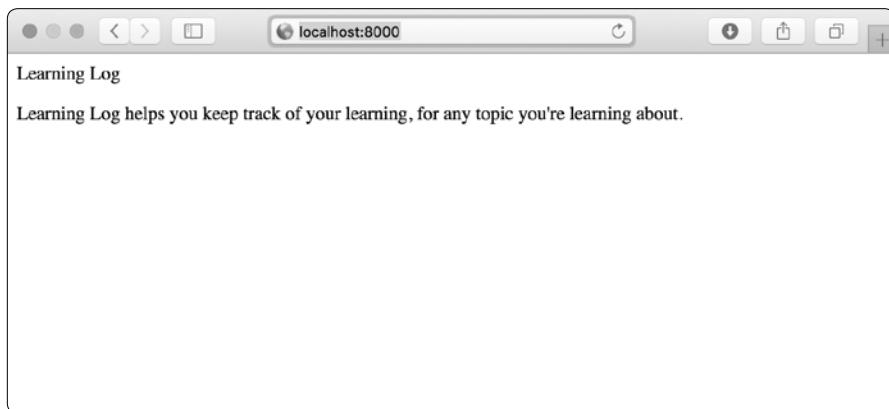


Figure 18-3: The home page for Learning Log

Although it might seem like a complicated process for creating one page, this separation between URLs, views, and templates works quite well. It allows you to think about each aspect of a project separately. In larger projects, it allows individuals working on the project to focus on the areas in which they’re strongest. For example, a database specialist can focus on the models, a programmer can focus on the view code, and a web designer can focus on the templates.

NOTE

You might see the following error message:

```
ModuleNotFoundError: No module named 'learning_logs.urls'
```

If you do, stop the development server by pressing CTRL-C in the terminal window where you issued the runserver command. Then reissue the command python manage.py runserver. You should be able to see the home page. Any time you run into an error like this, try stopping and restarting the server.

TRY IT YOURSELF

18-5. Meal Planner: Consider an app that helps people plan their meals throughout the week. Make a new folder called `meal_planner`, and start a new Django project inside this folder. Then make a new app called `meal_plans`. Make a simple home page for this project.

18-6. Pizzeria Home Page: Add a home page to the Pizzeria project you started in Exercise 18-4 (page 394).

Building Additional Pages

Now that we’ve established a routine for building a page, we can start to build out the Learning Log project. We’ll build two pages that display data: a page that lists all topics and a page that shows all the entries for a particular topic. For each page, we’ll specify a URL pattern, write a view function, and write a template. But before we do this, we’ll create a base template that all templates in the project can inherit from.

Template Inheritance

When building a website, some elements will always need to be repeated on each page. Rather than writing these elements directly into each page, you can write a base template containing the repeated elements and then have each page inherit from the base. This approach lets you focus on developing the unique aspects of each page and makes it much easier to change the overall look and feel of the project.

The Parent Template

We'll create a template called `base.html` in the same directory as `index.html`. This file will contain elements common to all pages; every other template will inherit from `base.html`. The only element we want to repeat on each page right now is the title at the top. Because we'll include this template on every page, let's make the title a link to the home page:

`base.html`

```
❶ <p>
❷   <a href="{% url 'learning_logs:index' %}">Learning Log</a>
</p>

❸ {% block content %}{% endblock content %}
```

The first part of this file creates a paragraph containing the name of the project, which also acts as a home page link. To generate a link, we use a *template tag*, which is indicated by braces and percent signs `{% %}`. A template tag generates information to be displayed on a page. Our template tag `{% url 'learning_logs:index' %}` generates a URL matching the URL pattern defined in `learning_logs/urls.py` with the name '`index`' ❶. In this example, `learning_logs` is the *namespace* and `index` is a uniquely named URL pattern in that namespace. The namespace comes from the value we assigned to `app_name` in the `learning_logs/urls.py` file.

In a simple HTML page, a link is surrounded by the *anchor* tag `<a>`:

```
<a href="link_url">link text</a>
```

Having the template tag generate the URL for us makes it much easier to keep our links up to date. We only need to change the URL pattern in `urls.py`, and Django will automatically insert the updated URL the next time the page is requested. Every page in our project will inherit from `base.html`, so from now on, every page will have a link back to the home page.

At ❷ we insert a pair of `block` tags. This block, named `content`, is a placeholder; the child template will define the kind of information that goes in the `content` block.

A child template doesn't have to define every block from its parent, so you can reserve space in parent templates for as many blocks as you like; the child template uses only as many as it requires.

NOTE

In Python code, we almost always use four spaces when we indent. Template files tend to have more levels of nesting than Python files, so it's common to use only two spaces for each indentation level. You just need to ensure that you're consistent.

The Child Template

Now we need to rewrite `index.html` to inherit from `base.html`. Add the following code to `index.html`:

`index.html`

```
❶ {% extends "learning_logs/base.html" %}

❷ {% block content %}
```

```
<p>Learning Log helps you keep track of your learning, for any topic you're  
learning about.</p>  
❸ {% endblock content %}
```

If you compare this to the original *index.html*, you can see that we've replaced the Learning Log title with the code for inheriting from a parent template ❶. A child template must have an `{% extends %}` tag on the first line to tell Django which parent template to inherit from. The file *base.html* is part of *learning_logs*, so we include *learning_logs* in the path to the parent template. This line pulls in everything contained in the *base.html* template and allows *index.html* to define what goes in the space reserved by the `content` block.

We define the content block at ❷ by inserting a `{% block %}` tag with the name `content`. Everything that we aren't inheriting from the parent template goes inside the `content` block. Here, that's the paragraph describing the Learning Log project. At ❸ we indicate that we're finished defining the content by using an `{% endblock content %}` tag. The `{% endblock %}` tag doesn't require a name, but if a template grows to contain multiple blocks, it can be helpful to know exactly which block is ending.

You can start to see the benefit of template inheritance: in a child template, we only need to include content that's unique to that page. This not only simplifies each template, but also makes it much easier to modify the site. To modify an element common to many pages, you only need to modify the parent template. Your changes are then carried over to every page that inherits from that template. In a project that includes tens or hundreds of pages, this structure can make it much easier and faster to improve your site.

NOTE

In a large project, it's common to have one parent template called `base.html` for the entire site and parent templates for each major section of the site. All the section templates inherit from `base.html`, and each page in the site inherits from a section template. This way you can easily modify the look and feel of the site as a whole, any section in the site, or any individual page. This configuration provides a very efficient way to work, and it encourages you to steadily update your site over time.

The Topics Page

Now that we have an efficient approach to building pages, we can focus on our next two pages: the general topics page and the page to display entries for a single topic. The topics page will show all topics that users have created, and it's the first page that will involve working with data.

The Topics URL Pattern

First, we define the URL for the topics page. It's common to choose a simple URL fragment that reflects the kind of information presented on the page.

We'll use the word *topics*, so the URL `http://localhost:8000/topics/` will return this page. Here's how we modify `learning_logs/urls.py`:

```
urls.py      """Defines URL patterns for learning_logs."""
--snip--
urlpatterns = [
    # Home page.
    path('', views.index, name='index'),
    # Page that shows all topics.
❶    path('topics/', views.topics, name='topics'),
]
```

We've simply added `topics/` into the string argument used for the home page URL ❶. When Django examines a requested URL, this pattern will match any URL that has the base URL followed by *topics*. You can include or omit a forward slash at the end, but there can't be anything else after the word *topics*, or the pattern won't match. Any request with a URL that matches this pattern will then be passed to the function `topics()` in `views.py`.

The Topics View

The `topics()` function needs to retrieve some data from the database and send it to the template. Here's what we need to add to `views.py`:

```
views.py    from django.shortcuts import render
❶ from .models import Topic

def index(request):
    --snip--

❷ def topics(request):
        """Show all topics."""
❸    topics = Topic.objects.order_by('date_added')
❹    context = {'topics': topics}
❺    return render(request, 'learning_logs/topics.html', context)
```

We first import the model associated with the data we need ❶. The `topics()` function needs one parameter: the `request` object Django received from the server ❷. At ❸ we query the database by asking for the `Topic` objects, sorted by the `date_added` attribute. We store the resulting queryset in `topics`.

At ❹ we define a context that we'll send to the template. A `context` is a dictionary in which the keys are names we'll use in the template to access the data, and the values are the data we need to send to the template. In this case, there's one key-value pair, which contains the set of topics we'll display on the page. When building a page that uses data, we pass the `context` variable to `render()` as well as the `request` object and the path to the template ❺.

The Topics Template

The template for the topics page receives the context dictionary, so the template can use the data that `topics()` provides. Make a file called `topics.html` in the same directory as `index.html`. Here's how we can display the topics in the template:

```
topics.html  {% extends "learning_logs/base.html" %}

{% block content %}

<p>Topics</p>

❶  <ul>
❷      {% for topic in topics %}
❸          <li>{{ topic }}</li>
❹          {% empty %}
              <li>No topics have been added yet.</li>
❺      {% endfor %}
❻  </ul>

{% endblock content %}
```

We use the `{% extends %}` tag to inherit from `base.html`, just as the index template does, and then open a content block. The body of this page contains a bulleted list of the topics that have been entered. In standard HTML, a bulleted list is called an *unordered list* and is indicated by the tags ``. We begin the bulleted list of topics at ❶.

At ❷ we have another template tag equivalent to a `for` loop, which loops through the list `topics` from the context dictionary. The code used in templates differs from Python in some important ways. Python uses indentation to indicate which lines of a `for` statement are part of a loop. In a template, every `for` loop needs an explicit `{% endfor %}` tag indicating where the end of the loop occurs. So in a template, you'll see loops written like this:

```
{% for item in list %}
    do something with each item
{% endfor %}
```

Inside the loop, we want to turn each topic into an item in the bulleted list. To print a variable in a template, wrap the variable name in double braces. The braces won't appear on the page; they just indicate to Django that we're using a template variable. So the code `{{ topic }}` at ❸ will be replaced by the value of `topic` on each pass through the loop. The HTML tag `` indicates a *list item*. Anything between these tags, inside a pair of `` tags, will appear as a bulleted item in the list.

At ❹ we use the `{% empty %}` template tag, which tells Django what to do if there are no items in the list. In this case, we print a message informing the user that no topics have been added yet. The last two lines close out the `for` loop ❺ and then close out the bulleted list ❻.

Now we need to modify the base template to include a link to the topics page. Add the following code to *base.html*:

```
base.html
<p>
❶ <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
❷ <a href="{% url 'learning_logs:topics' %}">Topics</a>
</p>

{% block content %}{% endblock content %}
```

We add a dash after the link to the home page ❶, and then add a link to the topics page using the `{% url %}` template tag again ❷. This line tells Django to generate a link matching the URL pattern with the name `'topics'` in *learning_logs/urls.py*.

Now when you refresh the home page in your browser, you'll see a *Topics* link. When you click the link, you'll see a page that looks similar to Figure 18-4.



Figure 18-4: The topics page

Individual Topic Pages

Next, we need to create a page that can focus on a single topic, showing the topic name and all the entries for that topic. We'll again define a new URL pattern, write a view, and create a template. We'll also modify the topics page so each item in the bulleted list links to its corresponding topic page.

The Topic URL Pattern

The URL pattern for the topic page is a little different than the prior URL patterns because it will use the topic's `id` attribute to indicate which topic was requested. For example, if the user wants to see the detail page for the Chess topic, where the `id` is 1, the URL will be `http://localhost:8000/topics/1/`.

Here's a pattern to match this URL, which you should place in *learning_logs/urls.py*:

```
urls.py
--snip--
urlpatterns = [
    --snip--
    # Detail page for a single topic.
    path('topics/<int:topic_id>', views.topic, name='topic'),
]
```

Let's examine the string '`topics/<int:topic_id>`' in this URL pattern. The first part of the string tells Django to look for URLs that have the word *topics* after the base URL. The second part of the string, `/<int:topic_id>/`, matches an integer between two forward slashes and stores the integer value in an argument called `topic_id`.

When Django finds a URL that matches this pattern, it calls the view function `topic()` with the value stored in `topic_id` as an argument. We'll use the value of `topic_id` to get the correct topic inside the function.

The Topic View

The `topic()` function needs to get the topic and all associated entries from the database, as shown here:

```
views.py
--snip--
❶ def topic(request, topic_id):
    """Show a single topic and all its entries."""
❷     topic = Topic.objects.get(id=topic_id)
❸     entries = topic.entry_set.order_by('-date_added')
❹     context = {'topic': topic, 'entries': entries}
❺     return render(request, 'learning_logs/topic.html', context)
```

This is the first view function that requires a parameter other than the `request` object. The function accepts the value captured by the expression `/<int:topic_id>/` and stores it in `topic_id` ❶. At ❷ we use `get()` to retrieve the topic, just as we did in the Django shell. At ❸ we get the entries associated with this topic, and we order them according to `date_added`. The minus sign in front of `date_added` sorts the results in reverse order, which will display the most recent entries first. We store the topic and entries in the context dictionary ❹ and send context to the template `topic.html` ❺.

NOTE

The code phrases at ❷ and ❸ are called queries, because they query the database for specific information. When you're writing queries like these in your own projects, it's helpful to try them out in the Django shell first. You'll get much quicker feedback in the shell than you will by writing a view and template, and then checking the results in a browser.

The Topic Template

The template needs to display the name of the topic and the entries. We also need to inform the user if no entries have been made yet for this topic.

```
topic.html    {% extends 'learning_logs/base.html' %}

            {% block content %}

❶    <p>Topic: {{ topic }}</p>

            <p>Entries:</p>
❷    <ul>
❸    {% for entry in entries %}
        <li>
❹        <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
❺        <p>{{ entry.text|linebreaks }}</p>
        </li>
❻    {% empty %}
        <li>There are no entries for this topic yet.</li>
    {% endfor %}
    </ul>

            {% endblock content %}
```

We extend *base.html*, as we do for all pages in the project. Next, we show the topic that's currently being displayed ❶, which is stored in the template variable `{{ topic }}`. The variable `topic` is available because it's included in the context dictionary. We then start a bulleted list to show each of the entries ❷ and loop through them as we did the topics earlier ❸.

Each bullet lists two pieces of information: the timestamp and the full text of each entry. For the timestamp ❹, we display the value of the attribute `date_added`. In Django templates, a vertical line (`|`) represents a template *filter*—a function that modifies the value in a template variable. The filter `date:'M d, Y H:i'` displays timestamps in the format *January 1, 2018 23:00*. The next line displays the full value of `text` rather than just the first 50 characters from `entry`. The filter `linebreaks` ❺ ensures that long text entries include line breaks in a format understood by browsers rather than showing a block of uninterrupted text. At ❻ we use the `{% empty %}` template tag to print a message informing the user that no entries have been made.

Links from the Topics Page

Before we look at the topic page in a browser, we need to modify the topics template so each topic links to the appropriate page. Here's the change you need to make to *topics.html*:

```
topics.html --snip--
            {% for topic in topics %}
                <li>
                    <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
                </li>
            {% empty %}
--snip--
```

We use the URL template tag to generate the proper link, based on the URL pattern in `learning_logs` with the name 'topic'. This URL pattern requires a `topic_id` argument, so we add the attribute `topic.id` to the URL template tag. Now each topic in the list of topics is a link to a topic page, such as `http://localhost:8000/topics/1/`.

When you refresh the topics page and click a topic, you should see a page that looks like Figure 18-5.

NOTE

There's a subtle but important difference between `topic.id` and `topic_id`. The expression `topic.id` examines a topic and retrieves the value of the corresponding ID. The variable `topic_id` is a reference to that ID in the code. If you run into errors when working with IDs, make sure you're using these expressions in the appropriate ways.

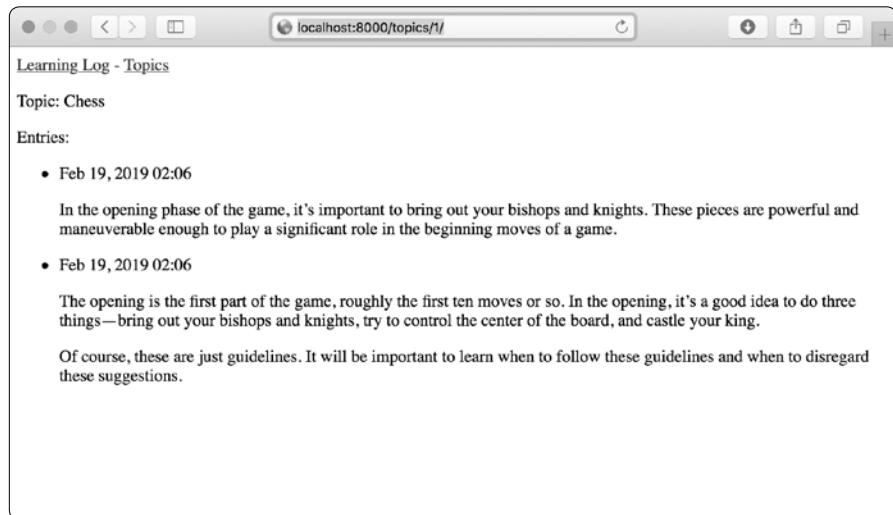


Figure 18-5: The detail page for a single topic, showing all entries for a topic

TRY IT YOURSELF

18-7. Template Documentation: Skim the Django template documentation at <https://docs.djangoproject.com/en/2.2/ref/templates/>. You can refer back to it when you're working on your own projects.

18-8. Pizzeria Pages: Add a page to the Pizzeria project from Exercise 18-6 (page 398) that shows the names of available pizzas. Then link each pizza name to a page displaying the pizza's toppings. Make sure you use template inheritance to build your pages efficiently.

Summary

In this chapter, you learned how to build simple web applications using the Django framework. You wrote a brief project specification, installed Django to a virtual environment, set up a project, and checked that the project was set up correctly. You set up an app and defined models to represent the data for your app. You learned about databases and how Django helps you migrate your database after you make a change to your models. You created a superuser for the admin site, and you used the admin site to enter some initial data.

You also explored the Django shell, which allows you to work with your project’s data in a terminal session. You learned to define URLs, create view functions, and write templates to make pages for your site. You also used template inheritance to simplify the structure of individual templates and make it easier to modify the site as the project evolves.

In Chapter 19, you’ll make intuitive, user-friendly pages that allow users to add new topics and entries and edit existing entries without going through the admin site. You’ll also add a user registration system, allowing users to create an account and make their own learning log. This is the heart of a web app—the ability to create something that any number of users can interact with.

19

USER ACCOUNTS



At the heart of a web application is the ability for any user, anywhere in the world, to register an account with your app and start using it. In this chapter, you'll build forms so users can add their own topics and entries, and edit existing entries. You'll also learn how Django guards against common attacks to form-based pages so you don't have to spend much time thinking about securing your apps.

You'll also implement a user authentication system. You'll build a registration page for users to create accounts, and then restrict access to certain pages to logged-in users only. Then you'll modify some of the view functions so users can only see their own data. You'll learn to keep your users' data safe and secure.

Allowing Users to Enter Data

Before we build an authentication system for creating accounts, we'll first add some pages that allow users to enter their own data. We'll give users the ability to add a new topic, add a new entry, and edit their previous entries.

Currently, only a superuser can enter data through the admin site. We don't want users to interact with the admin site, so we'll use Django's form-building tools to build pages that allow users to enter data.

Adding New Topics

Let's start by allowing users to add a new topic. Adding a form-based page works in much the same way as the pages we've already built: we define a URL, write a view function, and write a template. The one major difference is the addition of a new module called *forms.py*, which will contain the forms.

The Topic ModelForm

Any page that lets a user enter and submit information on a web page is a *form*, even if it doesn't look like one. When users enter information, we need to *validate* that the information provided is the right kind of data and is not malicious, such as code to interrupt our server. We then need to process and save valid information to the appropriate place in the database. Django automates much of this work.

The simplest way to build a form in Django is to use a *ModelForm*, which uses the information from the models we defined in Chapter 18 to automatically build a form. Write your first form in the file *forms.py*, which you should create in the same directory as *models.py*:

```
forms.py    from django import forms
              from .models import Topic

❶ class TopicForm(forms.ModelForm):
    class Meta:
        model = Topic
        fields = ['text']
❷        labels = {'text': ''}
```

We first import the *forms* module and the model we'll work with, called *Topic*. At ❶ we define a class called *TopicForm*, which inherits from *forms.ModelForm*.

The simplest version of a *ModelForm* consists of a nested *Meta* class telling Django which model to base the form on and which fields to include in the form. At ❷ we build a form from the *Topic* model and include only the *text* field ❸. The code at ❹ tells Django not to generate a label for the *text* field.

The new_topic URL

The URL for a new page should be short and descriptive. When the user wants to add a new topic, we'll send them to `http://localhost:8000/new_topic/`. Here's the URL pattern for the `new_topic` page, which you add to `learning_logs/urls.py`:

```
urls.py
--snip--
urlpatterns = [
    --snip--
    # Page for adding a new topic
    path('new_topic/', views.new_topic, name='new_topic'),
]
```

This URL pattern sends requests to the view function `new_topic()`, which we'll write next.

The new_topic() View Function

The `new_topic()` function needs to handle two different situations: initial requests for the `new_topic` page (in which case it should show a blank form) and the processing of any data submitted in the form. After data from a submitted form is processed, it needs to redirect the user back to the `topics` page:

```
views.py
from django.shortcuts import render, redirect

from .models import Topic
from .forms import TopicForm

--snip--
def new_topic(request):
    """Add a new topic."""
    if request.method != 'POST':
        # No data submitted; create a blank form.
        form = TopicForm()
    else:
        # POST data submitted; process data.
        form = TopicForm(data=request.POST)
        if form.is_valid():
            form.save()
    return redirect('learning_logs:topics')

    # Display a blank or invalid form.
    context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
```

We import the function `redirect`, which we'll use to redirect the user back to the `topics` page after they submit their topic. The `redirect()` function takes in the name of a view and redirects the user to that view. We also import the form we just wrote, `TopicForm`.

GET and POST Requests

The two main types of request you'll use when building web apps are GET requests and POST requests. You use *GET* requests for pages that only read data from the server. You usually use *POST* requests when the user needs to submit information through a form. We'll be specifying the POST method for processing all of our forms. (A few other kinds of requests exist, but we won't use them in this project.)

The `new_topic()` function takes in the `request` object as a parameter. When the user initially requests this page, their browser will send a GET request. Once the user has filled out and submitted the form, their browser will submit a POST request. Depending on the request, we'll know whether the user is requesting a blank form (a GET request) or asking us to process a completed form (a POST request).

The test at ❶ determines whether the request method is GET or POST. If the request method isn't POST, the request is probably GET, so we need to return a blank form (if it's another kind of request, it's still safe to return a blank form). We make an instance of `TopicForm` ❷, assign it to the variable `form`, and send the form to the template in the context dictionary ❸. Because we included no arguments when instantiating `TopicForm`, Django creates a blank form that the user can fill out.

If the request method is POST, the `else` block runs and processes the data submitted in the form. We make an instance of `TopicForm` ❹ and pass it the data entered by the user, stored in `request.POST`. The `form` object that's returned contains the information submitted by the user.

We can't save the submitted information in the database until we've checked that it's valid ❺. The `is_valid()` method checks that all required fields have been filled in (all fields in a form are required by default) and that the data entered matches the field types expected—for example, that the length of text is less than 200 characters, as we specified in `models.py` in Chapter 18. This automatic validation saves us a lot of work. If everything is valid, we can call `save()` ❻, which writes the data from the form to the database.

Once we've saved the data, we can leave this page. We use `redirect()` to redirect the user's browser to the topics page, where the user should see the topic they just entered in the list of topics.

The context variable is defined at the end of the view function, and the page is rendered using the template `new_topic.html`, which we'll create next. This code is placed outside of any `if` block; it will run if a blank form was created, and it will run if a submitted form is determined to be invalid. An invalid form will include some default error messages to help the user submit acceptable data.

The `new_topic` Template

Now we'll make a new template called `new_topic.html` to display the form we just created.

```
new_topic.html    {% extends "learning_logs/base.html" %}

    {% block content %}
        <p>Add a new topic:</p>

❶    <form action="{% url 'learning_logs:new_topic' %}" method='post'>
❷        {% csrf_token %}
❸        {{ form.as_p }}
❹        <button name="submit">Add topic</button>
    </form>

    {% endblock content %}
```

This template extends *base.html*, so it has the same base structure as the rest of the pages in Learning Log. At ❶ we define an HTML form. The `action` argument tells the browser where to send the data submitted in the form; in this case, we send it back to the view function `new_topic()`. The `method` argument tells the browser to submit the data as a POST request.

Django uses the template tag `{% csrf_token %}` ❷ to prevent attackers from using the form to gain unauthorized access to the server (this kind of attack is called a *cross-site request forgery*). At ❸ we display the form; here you see how simple Django can make certain tasks, such as displaying a form. We only need to include the template variable `{{ form.as_p }}` for Django to create all the fields necessary to display the form automatically. The `as_p` modifier tells Django to render all the form elements in paragraph format, as a simple way to display the form neatly.

Django doesn't create a submit button for forms, so we define one at ❹.

Linking to the new_topic Page

Next, we include a link to the `new_topic` page on the `topics` page:

```
topics.html      {% extends "learning_logs/base.html" %}

    {% block content %}

        <p>Topics</p>

        <ul>
            --snip--
        </ul>

        <a href="{% url 'learning_logs:new_topic' %}">Add a new topic</a>

    {% endblock content %}
```

Place the link after the list of existing topics. Figure 19-1 shows the resulting form. Use the form to add a few new topics of your own.



Figure 19-1: The page for adding a new topic

Adding New Entries

Now that the user can add a new topic, they'll want to add new entries too. We'll again define a URL, write a view function and a template, and link to the page. But first, we'll add another class to *forms.py*.

The Entry ModelForm

We need to create a form associated with the `Entry` model but this time with a bit more customization than `TopicForm`:

```
forms.py    from django import forms
              from .models import Topic, Entry

              class TopicForm(forms.ModelForm):
                  --snip--

              class EntryForm(forms.ModelForm):
                  class Meta:
                      model = Entry
                      fields = ['text']
                      labels = {'text': 'Entry:'}
①                      widgets = {'text': forms.Textarea(attrs={'cols': 80})}
```

We update the `import` statement to include `Entry` as well as `Topic`. We make a new class called `EntryForm` that inherits from `forms.ModelForm`. The `EntryForm` class has a nested `Meta` class listing the model it's based on and the field to include in the form. We again give the field 'text' a blank label ①.

At ② we include the `widgets` attribute. A *widget* is an HTML form element, such as a single-line text box, multi-line text area, or drop-down list. By including the `widgets` attribute, you can override Django's default widget choices. By telling Django to use a `forms.Textarea` element, we're customizing

the input widget for the field 'text' so the text area will be 80 columns wide instead of the default 40. This gives users enough room to write a meaningful entry.

The new_entry URL

New entries must be associated with a particular topic, so we need to include a `topic_id` argument in the URL for adding a new entry. Here's the URL, which you add to `learning_logs/urls.py`:

```
urls.py
--snip--
urlpatterns = [
    --snip--
    # Page for adding a new entry
    path('new_entry/<int:topic_id>', views.new_entry, name='new_entry'),
]
```

This URL pattern matches any URL with the form `http://localhost:8000/new_entry/id/`, where `id` is a number matching the topic ID. The code `<int:topic_id>` captures a numerical value and assigns it to the variable `topic_id`. When a URL matching this pattern is requested, Django sends the request and the topic's ID to the `new_entry()` view function.

The new_entry() View Function

The view function for `new_entry` is much like the function for adding a new topic. Add the following code to your `views.py` file:

```
views.py
from django.shortcuts import render, redirect

from .models import Topic
from .forms import TopicForm, EntryForm

--snip--
def new_entry(request, topic_id):
    """Add a new entry for a particular topic."""
❶    topic = Topic.objects.get(id=topic_id)

❷    if request.method != 'POST':
        # No data submitted; create a blank form.
❸    form = EntryForm()
    else:
        # POST data submitted; process data.
❹    form = EntryForm(data=request.POST)
        if form.is_valid():
❺        new_entry = form.save(commit=False)
❻        new_entry.topic = topic
            new_entry.save()
❾    return redirect('learning_logs:topic', topic_id=topic_id)

    # Display a blank or invalid form.
    context = {'topic': topic, 'form': form}
    return render(request, 'learning_logs/new_entry.html', context)
```

We update the import statement to include the `EntryForm` we just made. The definition of `new_entry()` has a `topic_id` parameter to store the value it receives from the URL. We'll need the topic to render the page and process the form's data, so we use `topic_id` to get the correct topic object at ❶.

At ❷ we check whether the request method is POST or GET. The `if` block executes if it's a GET request, and we create a blank instance of `EntryForm` ❸.

If the request method is POST, we process the data by making an instance of `EntryForm`, populated with the POST data from the request object ❹. We then check whether the form is valid. If it is, we need to set the entry object's `topic` attribute before saving it to the database. When we call `save()`, we include the argument `commit=False` ❺ to tell Django to create a new entry object and assign it to `new_entry` without saving it to the database yet. We set the `topic` attribute of `new_entry` to the topic we pulled from the database at the beginning of the function ❻. Then we call `save()` with no arguments, saving the entry to the database with the correct associated topic.

The `redirect()` call at ❼ requires two arguments—the name of the view we want to redirect to and the argument that view function requires. Here, we're redirecting to `topic()`, which needs the argument `topic_id`. This view then renders the topic page that the user made an entry for, and they should see their new entry in the list of entries.

At the end of the function, we create a context dictionary and render the page using the `new_entry.html` template. This code will execute for a blank form or for a submitted form that is evaluated as invalid.

The `new_entry` Template

As you can see in the following code, the template for `new_entry` is similar to the template for `new_topic`:

```
new_entry.html
{% extends "learning_logs/base.html" %}

{% block content %}

❶ <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

❷ <p>Add a new entry:</p>
<form action="{% url 'learning_logs:new_entry' topic.id %}" method='post'>
    {% csrf_token %}
    {{ form.as_p }}
    <button name='submit'>Add entry</button>
</form>

{% endblock content %}
```

We show the topic at the top of the page ❶, so the user can see which topic they're adding an entry to. The topic also acts as a link back to the main page for that topic.

The form's `action` argument includes the `topic_id` value in the URL, so the view function can associate the new entry with the correct topic ❷. Other than that, this template looks just like `new_topic.html`.

Linking to the `new_entry` Page

Next, we need to include a link to the `new_entry` page from each topic page in the topic template:

```
topic.html
{%
    extends "learning_logs/base.html"
}

{% block content %}

<p>Topic: {{ topic }}</p>

<p>Entries:</p>
<p>
    <a href="{% url 'learning_logs:new_entry' topic.id %}">Add new entry</a>
</p>

<ul>
--snip--
</ul>

{% endblock content %}
```

We place the link to add entries just before showing the entries, because adding a new entry will be the most common action on this page. Figure 19-2 shows the `new_entry` page. Now users can add new topics and as many entries as they want for each topic. Try out the `new_entry` page by adding a few entries to some of the topics you've created.

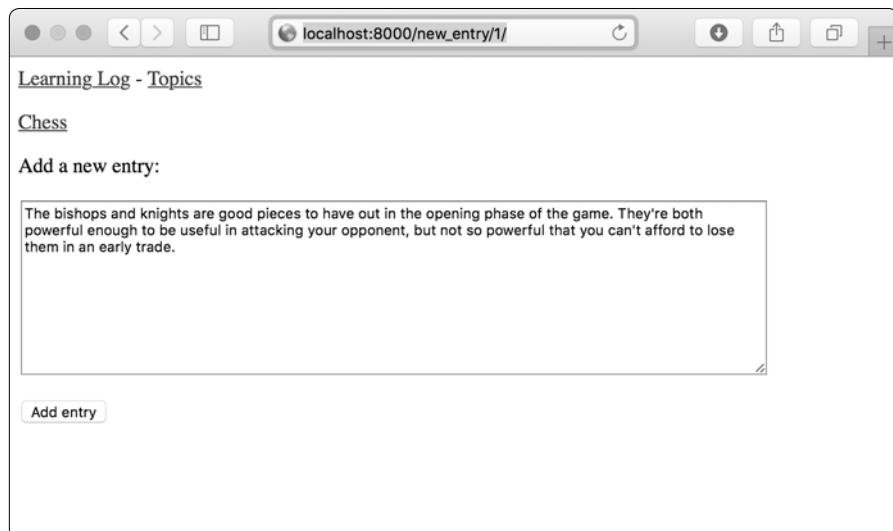


Figure 19-2: The `new_entry` page

Editing Entries

Now we'll make a page so users can edit the entries they've added.

The `edit_entry` URL

The URL for the page needs to pass the ID of the entry to be edited. Here's `learning_logs/urls.py`:

```
urls.py
--snip--
urlpatterns = [
    --snip--
    # Page for editing an entry.
    path('edit_entry/<int:entry_id>', views.edit_entry, name='edit_entry'),
]
```

The ID passed in the URL (for example, `http://localhost:8000/edit_entry/1/`) is stored in the parameter `entry_id`. The URL pattern sends requests that match this format to the view function `edit_entry()`.

The `edit_entry()` View Function

When the `edit_entry` page receives a GET request, the `edit_entry()` function returns a form for editing the entry. When the page receives a POST request with revised entry text, it saves the modified text into the database:

```
views.py
from django.shortcuts import render, redirect

from .models import Topic, Entry
from .forms import TopicForm, EntryForm
--snip--

def edit_entry(request, entry_id):
    """Edit an existing entry."""
①    entry = Entry.objects.get(id=entry_id)
    topic = entry.topic

    if request.method != 'POST':
        # Initial request; pre-fill form with the current entry.
②        form = EntryForm(instance=entry)
    else:
        # POST data submitted; process data.
③        form = EntryForm(instance=entry, data=request.POST)
        if form.is_valid():
            form.save()
④            return redirect('learning_logs:topic', topic_id=topic.id)

    context = {'entry': entry, 'topic': topic, 'form': form}
    return render(request, 'learning_logs/edit_entry.html', context)
```

We first import the `Entry` model. At ❶ we get the entry object that the user wants to edit and the topic associated with this entry. In the `if` block, which runs for a GET request, we make an instance of `EntryForm` with the argument `instance=entry` ❷. This argument tells Django to create the form prefilled with information from the existing entry object. The user will see their existing data and be able to edit that data.

When processing a POST request, we pass the `instance=entry` argument and the `data=request.POST` argument ❸. These arguments tell Django to create a form instance based on the information associated with the existing entry object, updated with any relevant data from `request.POST`. We then check whether the form is valid; if it is, we call `save()` with no arguments because the entry is already associated with the correct topic ❹. We then redirect to the topic page, where the user should see the updated version of the entry they edited ❺.

If we're showing an initial form for editing the entry or if the submitted form is invalid, we create the context dictionary and render the page using the `edit_entry.html` template.

The `edit_entry` Template

Next, we create an `edit_entry.html` template, which is similar to `new_entry.html`:

```
edit_entry.html
{%
    extends "learning_logs/base.html"
    %}

    {% block content %}

        <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

        <p>Edit entry:</p>

    ❶    <form action="{% url 'learning_logs:edit_entry' entry.id %}" method="post">
        {% csrf_token %}
        {{ form.as_p }}
    ❷        <button name="submit">Save changes</button>
    </form>

    {% endblock content %}
```

At ❶ the `action` argument sends the form back to the `edit_entry()` function for processing. We include the entry ID as an argument in the `{% url %}` tag, so the view function can modify the correct entry object. We label the submit button as *Save changes* to remind the user they're saving edits, not creating a new entry ❷.

Linking to the edit_entry Page

Now we need to include a link to the edit_entry page for each entry on the topic page:

topic.html

```
--snip--  
    {% for entry in entries %}  
        <li>  
            <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>  
            <p>{{ entry.text|linebreaks }}</p>  
            <p>  
                <a href="{% url 'learning_logs:edit_entry' entry.id %}">Edit entry</a>  
            </p>  
        </li>  
    --snip--
```

We include the edit link after each entry's date and text has been displayed. We use the `{% url %}` template tag to determine the URL for the named URL pattern `edit_entry`, along with the ID attribute of the current entry in the loop (`entry.id`). The link text *Edit entry* appears after each entry on the page. Figure 19-3 shows what the topic page looks like with these links.

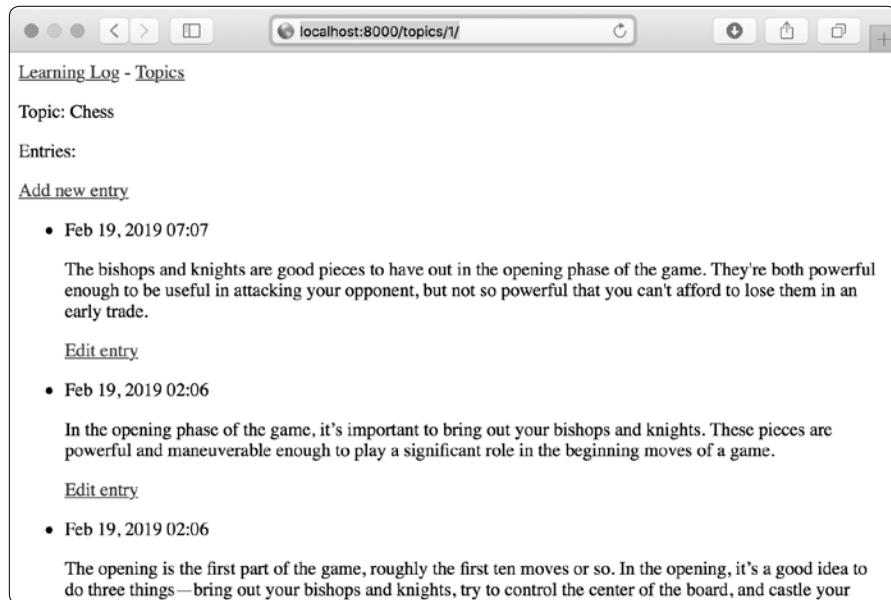


Figure 19-3: Each entry now has a link for editing that entry.

Learning Log now has most of the functionality it needs. Users can add topics and entries, and read through any set of entries they want. In the next section, we'll implement a user registration system so anyone can make an account with Learning Log and create their own set of topics and entries.

TRY IT YOURSELF

19-1. Blog: Start a new Django project called *Blog*. Create an app called *blogs* in the project and a model called *BlogPost*. The model should have fields like *title*, *text*, and *date_added*. Create a superuser for the project, and use the admin site to make a couple of short posts. Make a home page that shows all posts in chronological order.

Create a form for making new posts and another for editing existing posts. Fill in your forms to make sure they work.

Setting Up User Accounts

In this section, we'll set up a user registration and authorization system so people can register an account and log in and out. We'll create a new app to contain all the functionality related to working with users. We'll use the default user authentication system included with Django to do as much of the work as possible. We'll also modify the *Topic* model slightly so every topic belongs to a certain user.

The *users* App

We'll start by creating a new app called *users*, using the `startapp` command:

```
(11_env)learning_log$ python manage.py startapp users
(11_env)learning_log$ ls
❶ db.sqlite3 learning_log learning_logs 11_env manage.py users
(11_env)learning_log$ ls users
❷ __init__.py admin.py apps.py migrations models.py tests.py views.py
```

This command makes a new directory called *users* ❶ with a structure identical to the *learning_logs* app ❷.

Adding users to *settings.py*

We need to add our new app to `INSTALLED_APPS` in *settings.py*, like so:

```
settings.py
--snip--
INSTALLED_APPS = [
    # My apps
    'learning_logs',
    'users',

    # Default django apps.
    --snip--
]
--snip--
```

Now Django will include the `users` app in the overall project.

Including the URLs from `users`

Next, we need to modify the root `urls.py` so it includes the URLs we'll write for the `users` app:

`urls.py`

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('users/', include('users.urls')),
    path('', include('learning_logs.urls')),
]
```

We add a line to include the file `urls.py` from `users`. This line will match any URL that starts with the word `users`, such as `http://localhost:8000/users/login/`.

The Login Page

We'll first implement a login page. We'll use the default `login` view Django provides, so the URL pattern for this app looks a little different. Make a new `urls.py` file in the directory `learning_log/users/`, and add the following to it:

`urls.py`

```
"""Defines URL patterns for users"""

from django.urls import path, include

❶ app_name = 'users'
urlpatterns = [
    # Include default auth urls.
❷     path('', include('django.contrib.auth.urls')),
]
```

We import the `path` function, and then import the `include` function so we can include some default authentication URLs that Django has defined. These default URLs include named URL patterns, such as `'login'` and `'logout'`. We set the variable `app_name` to `'users'` so Django can distinguish these URLs from URLs belonging to other apps ❶. Even default URLs provided by Django, when included in the `users` app's `urls.py` file, will be accessible through the `users` namespace.

The login page's pattern matches the URL `http://localhost:8000/users/login/` ❷. When Django reads this URL, the word `users` tells Django to look in `users/urls.py`, and `login` tells it to send requests to Django's default `login` view.

The login Template

When the user requests the login page, Django will use a default view function, but we still need to provide a template for the page. The default

authentication views look for templates inside a folder called *registration*, so we'll need to make that folder. Inside the *learning_log/users* directory, make a directory called *templates*; inside that, make another directory called *registration*. Here's the *login.html* template, which you should save in *learning_log/users/templates/registration*:

```
login.html  {% extends "learning_logs/base.html" %}

    {% block content %}

        ❶  {% if form.errors %}
            <p>Your username and password didn't match. Please try again.</p>
            {% endif %}

        ❷  <form method="post" action="{% url 'users:login' %}">
            {% csrf_token %}
        ❸  {{ form.as_p }}

        ❹  <button name="submit">Log in</button>
        ❺  <input type="hidden" name="next"
            value="{% url 'learning_logs:index' %}" />
        </form>

    {% endblock content %}
```

This template extends *base.html* to ensure that the login page will have the same look and feel as the rest of the site. Note that a template in one app can inherit from a template in another app.

If the form's *errors* attribute is set, we display an error message ❶, reporting that the username and password combination don't match anything stored in the database.

We want the login view to process the form, so we set the *action* argument as the URL of the login page ❷. The login view sends a form to the template, and it's up to us to display the form ❸ and add a submit button ❹. At ❺ we include a hidden form element, 'next'; the *value* argument tells Django where to redirect the user after they've logged in successfully. In this case, we send the user back to the home page.

Linking to the Login Page

Let's add the login link to *base.html* so it appears on every page. We don't want the link to display when the user is already logged in, so we nest it inside an *{% if %}* tag:

```
base.html  <p>
            <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
            <a href="{% url 'learning_logs:topics' %}">Topics</a> -
        ❶  {% if user.is_authenticated %}
            Hello, {{ user.username }}.
        ❷  {% else %}
            <a href="{% url 'users:login' %}">Log in</a>
        ❸  {% endif %}
```

```
</p>  
{%- block content %}{% endblock content %}
```

In Django's authentication system, every template has a `user` variable available, which always has an `is_authenticated` attribute set: the attribute is `True` if the user is logged in and `False` if they aren't. This attribute allows you to display one message to authenticated users and another to unauthenticated users.

Here we display a greeting to users currently logged in ❶. Authenticated users have an additional `username` attribute set, which we use to personalize the greeting and remind the user they're logged in ❷. At ❸ we display a link to the login page for users who haven't been authenticated.

Using the Login Page

We've already set up a user account, so let's log in to see if the page works. Go to `http://localhost:8000/admin/`. If you're still logged in as an admin, look for a logout link in the header and click it.

When you're logged out, go to `http://localhost:8000/users/login/`. You should see a login page similar to the one shown in Figure 19-4. Enter the username and password you set up earlier, and you should be brought back to the index page. The header on the home page should display a greeting personalized with your username.

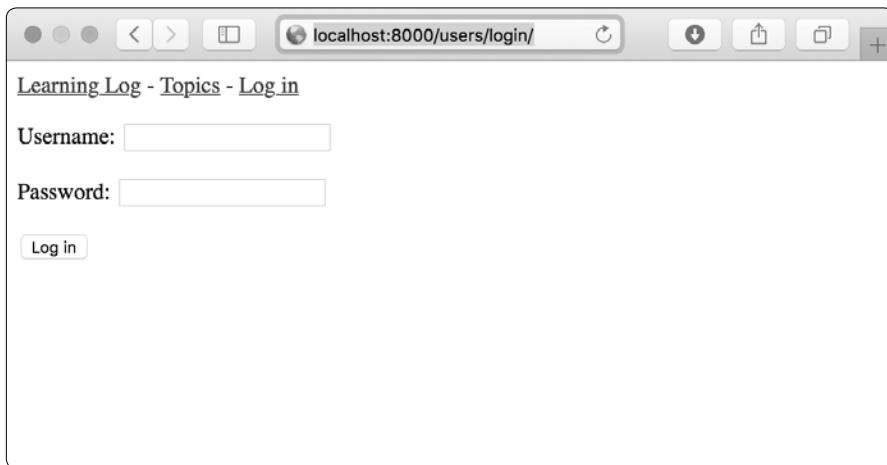


Figure 19-4: The login page

Logging Out

Now we need to provide a way for users to log out. We'll put a link in `base.html` that logs out users; when they click this link, they'll go to a page confirming that they've been logged out.

Adding a Logout Link to base.html

We'll add the link for logging out to *base.html* so it's available on every page. We'll include it in the `{% if user.is_authenticated %}` portion so only users who are already logged in can see it:

base.html

```
--snip--  
  {% if user.is_authenticated %}  
    Hello, {{ user.username }}.  
    <a href="{% url 'users:logout' %}">Log out</a>  
  {% else %}  
    --snip--
```

The default named URL pattern for logging out is simply 'logout'.

The Logout Confirmation Page

Users will want to know that they've successfully logged out, so the default logout view renders the page using the template *logged_out.html*, which we'll create now. Here's a simple page confirming that the user has been logged out. Save this file in *templates/registration*, the same place where you saved *login.html*:

logged_out.html

```
{% extends "learning_logs/base.html" %}  
  
{% block content %}  
  <p>You have been logged out. Thank you for visiting!</p>  
{% endblock content %}
```

We don't need anything else on this page, because *base.html* provides links back to the home page and the login page if the user wants to go back to either page.

Figure 19-5 shows the logged out page as it appears to a user who has just clicked the *Log out* link. The styling is minimal because we're focusing on building a site that works properly. When the required set of features works, we'll style the site to look more professional.

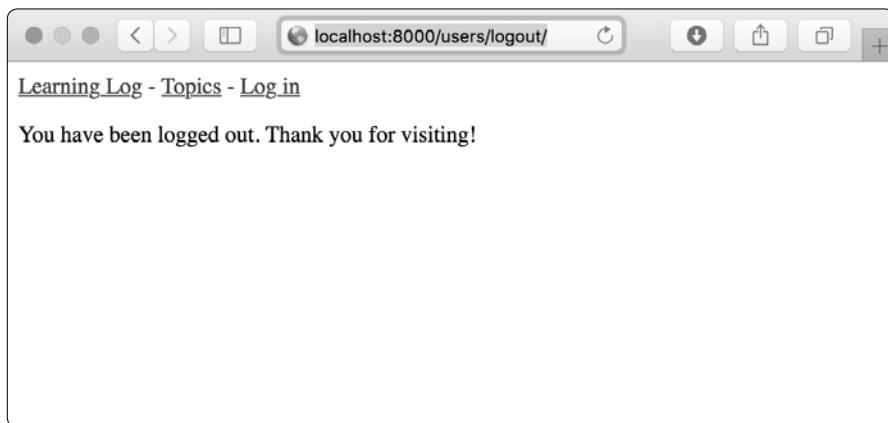


Figure 19-5: The logged out page confirms that a user has successfully logged out.

The Registration Page

Next, we'll build a page so new users can register. We'll use Django's default `UserCreationForm` but write our own view function and template.

The register URL

The following code provides the URL pattern for the registration page, again in `users/urls.py`:

`urls.py`

```
"""Defines URL patterns for users"""

from django.urls import path, include
from . import views

app_name = 'users'
urlpatterns = [
    # Include default auth urls.
    path('', include('django.contrib.auth.urls')),
    # Registration page.
    path('register/', views.register, name='register'),
]
```

We import the `views` module from `users`, which we need because we're writing our own view for the registration page. The pattern for the registration page matches the URL `http://localhost:8000/users/register/` and sends requests to the `register()` function we're about to write.

The register() View Function

The `register()` view function needs to display a blank registration form when the registration page is first requested and then process completed registration forms when they're submitted. When a registration is successful, the function also needs to log in the new user. Add the following code to `users/views.py`:

`views.py`

```
from django.shortcuts import render, redirect
from django.contrib.auth import login
from django.contrib.auth.forms import UserCreationForm

def register(request):
    """Register a new user."""
    if request.method != 'POST':
        # Display blank registration form.
    ❶    form = UserCreationForm()
    else:
        # Process completed form.
    ❷    form = UserCreationForm(data=request.POST)

    ❸    if form.is_valid():
    ❹        new_user = form.save()
        # Log the user in and then redirect to home page.
```

```
❸     login(request, new_user)
❹     return redirect('learning_logs:index')

# Display a blank or invalid form.
context = {'form': form}
return render(request, 'registration/register.html', context)
```

We import the `render()` and `redirect()` functions. Then we import the `login()` function to log the user in if their registration information is correct. We also import the default `UserCreationForm`. In the `register()` function, we check whether or not we're responding to a POST request. If we're not, we make an instance of `UserCreationForm` with no initial data ❶.

If we're responding to a POST request, we make an instance of `UserCreationForm` based on the submitted data ❷. We check that the data is valid ❸—in this case, that the username has the appropriate characters, the passwords match, and the user isn't trying to do anything malicious in their submission.

If the submitted data is valid, we call the form's `save()` method to save the username and the hash of the password to the database ❹. The `save()` method returns the newly created user object, which we assign to `new_user`. When the user's information is saved, we log them in by calling the `login()` function with the `request` and `new_user` objects ❺, which creates a valid session for the new user. Finally, we redirect the user to the home page ❻, where a personalized greeting in the header tells them their registration was successful.

At the end of the function we render the page, which will either be a blank form or a submitted form that is invalid.

The register Template

Now create a template for the registration page, which will be similar to the login page. Be sure to save it in the same directory as `login.html`:

```
register.html  {% extends "learning_logs/base.html" %}

{% block content %}

<form method="post" action="{% url 'users:register' %}">
    {% csrf_token %}
    {{ form.as_p }}

    <button name="submit">Register</button>
    <input type="hidden" name="next" value="{% url 'learning_logs:index' %}" />
</form>

{% endblock content %}
```

We use the `as_p` method again so Django will display all the fields in the form appropriately, including any error messages if the form isn't filled out correctly.

Linking to the Registration Page

Next, we'll add the code to show the registration page link to any user who isn't currently logged in:

base.html

```
--snip--  
  {% if user.is_authenticated %}  
    Hello, {{ user.username }}.  
    <a href="{% url 'users:logout' %}">Log out</a>  
  {% else %}  
    <a href="{% url 'users:register' %}">Register</a> -  
    <a href="{% url 'users:login' %}">Log in</a>  
  {% endif %}  
--snip--
```

Now users who are logged in see a personalized greeting and a logout link. Users who aren't logged in see a registration page link and a login link. Try out the registration page by making several user accounts with different usernames.

In the next section, we'll restrict some of the pages so they're available only to registered users, and we'll make sure every topic belongs to a specific user.

NOTE

The registration system we've set up allows anyone to make any number of accounts for Learning Log. But some systems require users to confirm their identity by sending a confirmation email the user must reply to. By doing so, the system generates fewer spam accounts than the simple system we're using here. However, when you're learning to build apps, it's perfectly appropriate to practice with a simple user registration system like the one we're using.

TRY IT YOURSELF

19-2. Blog Accounts: Add a user authentication and registration system to the Blog project you started in Exercise 19-1 (page 421). Make sure logged-in users see their username somewhere on the screen and unregistered users see a link to the registration page.

Allowing Users to Own Their Data

Users should be able to enter data exclusive to them, so we'll create a system to figure out which data belongs to which user. Then we'll restrict access to certain pages so users can work with only their own data.

We'll modify the Topic model so every topic belongs to a specific user. This will also take care of entries, because every entry belongs to a specific topic. We'll start by restricting access to certain pages.

Restricting Access with `@login_required`

Django makes it easy to restrict access to certain pages to logged-in users through the `@login_required` decorator. A *decorator* is a directive placed just before a function definition that Python applies to the function before it runs, to alter how the function code behaves. Let's look at an example.

Restricting Access to the Topics Page

Each topic will be owned by a user, so only registered users can request the topics page. Add the following code to `learning_logs/views.py`:

`views.py`

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required

from .models import Topic, Entry
--snip--

@login_required
def topics(request):
    """Show all topics."""
    --snip--
```

We first import the `login_required()` function. We apply `login_required()` as a decorator to the `topics()` view function by prepending `login_required` with the `@` symbol. As a result, Python knows to run the code in `login_required()` before the code in `topics()`.

The code in `login_required()` checks whether a user is logged in, and Django runs the code in `topics()` only if they are. If the user isn't logged in, they're redirected to the login page.

To make this redirect work, we need to modify `settings.py` so Django knows where to find the login page. Add the following at the very end of `settings.py`:

`settings.py`

```
--snip--

# My settings
LOGIN_URL = 'users:login'
```

Now when an unauthenticated user requests a page protected by the `@login_required` decorator, Django will send the user to the URL defined by `LOGIN_URL` in `settings.py`.

You can test this setting by logging out of any user accounts and going to the home page. Click the **Topics** link, which should redirect you to the login page. Then log in to any of your accounts, and from the home page click the **Topics** link again. You should be able to access the topics page.

Restricting Access Throughout Learning Log

Django makes it easy to restrict access to pages, but you have to decide which pages to protect. It's best to think about which pages need to be unrestricted first, and then restrict all the other pages in the project. You can easily correct overrestricting access, and it's less dangerous than leaving sensitive pages unrestricted.

In Learning Log, we'll keep the home page and the registration page unrestricted. We'll restrict access to every other page.

Here's *learning_logs/views.py* with `@login_required` decorators applied to every view except `index()`:

```
views.py
--snip--
@login_required
def topics(request):
    --snip--

@login_required
def topic(request, topic_id):
    --snip--

@login_required
def new_topic(request):
    --snip--

@login_required
def new_entry(request, topic_id):
    --snip--

@login_required
def edit_entry(request, entry_id):
    --snip--
```

Try accessing each of these pages while logged out: you'll be redirected back to the login page. You'll also be unable to click links to pages such as `new_topic`. But if you enter the URL `http://localhost:8000/new_topic/`, you'll be redirected to the login page. You should restrict access to any URL that's publicly accessible and relates to private user data.

Connecting Data to Certain Users

Next, we need to connect the data to the user who submitted it. We need to connect only the data highest in the hierarchy to a user, and the lower-level data will follow. For example, in Learning Log, topics are the highest level of data in the app, and all entries are connected to a topic. As long as each topic belongs to a specific user, we can trace the ownership of each entry in the database.

We'll modify the `Topic` model by adding a foreign key relationship to a user. We'll then have to migrate the database. Finally, we'll modify some of the views so they only show the data associated with the currently logged in user.

Modifying the Topic Model

The modification to `models.py` is just two lines:

`models.py`

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
    """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(auto_now_add=True)
    owner = models.ForeignKey(User, on_delete=models.CASCADE)

    def __str__(self):
        """Return a string representation of the model."""
        return self.text

class Entry(models.Model):
    --snip--
```

We import the `User` model from `django.contrib.auth`. Then we add an `owner` field to `Topic`, which establishes a foreign key relationship to the `User` model. If a user is deleted, all the topics associated with that user will be deleted as well.

Identifying Existing Users

When we migrate the database, Django will modify the database so it can store a connection between each topic and a user. To make the migration, Django needs to know which user to associate with each existing topic. The simplest approach is to start by giving all existing topics to one user—for example, the superuser. But first we need to know that user's ID.

Let's look at the IDs of all users created so far. Start a Django shell session and issue the following commands:

```
(ll_env)learning_log$ python manage.py shell
❶ >>> from django.contrib.auth.models import User
❷ >>> User.objects.all()
<QuerySet [User: ll_admin, User: eric, User: willie]>
❸ >>> for user in User.objects.all():
...     print(user.username, user.id)
...
ll_admin 1
eric 2
willie 3
>>>
```

At ❶ we import the `User` model into the shell session. We then look at all the users that have been created so far ❷. The output shows three users: `ll_admin`, `eric`, and `willie`.

At ❸ we loop through the list of users and print each user's username and ID. When Django asks which user to associate the existing topics with, we'll use one of these ID values.

Migrating the Database

Now that we know the IDs, we can migrate the database. When we do this, Python will ask us to connect the `Topic` model to a particular owner temporarily or to add a default to our `models.py` file to tell it what to do. Choose option 1:

```
❶ (11_env)learning_log$ python manage.py makemigrations learning_logs
❷ You are trying to add a non-nullable field 'owner' to topic without a default;
   we can't do that (the database needs something to populate existing rows).
❸ Please select a fix:
   1) Provide a one-off default now (will be set on all existing rows with a
      null value for this column)
   2) Quit, and let me add a default in models.py
❹ Select an option: 1
❺ Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so you can do
e.g. timezone.now
Type 'exit' to exit this prompt
❻ >>> 1
Migrations for 'learning_logs':
    learning_logs/migrations/0003_topic_owner.py
    - Add field owner to topic
(11_env)learning_log$
```

We start by issuing the `makemigrations` command ❶. In the output at ❷, Django indicates that we're trying to add a required (non-nullable) field to an existing model (`topic`) with no default value specified. Django gives us two options at ❸: we can provide a default right now, or we can quit and add a default value in `models.py`. At ❹ we've chosen the first option. Django then asks us to enter the default value ❻.

To associate all existing topics with the original admin user, `ll_admin`, I entered the user ID of 1 at ❻. You can use the ID of any user you've created; it doesn't have to be a superuser. Django then migrates the database using this value and generates the migration file `0003_topic_owner.py`, which adds the field `owner` to the `Topic` model.

Now we can execute the migration. Enter the following in an active virtual environment:

```
(11_env)learning_log$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
Running migrations:
❶  Applying learning_logs.0003_topic_owner... OK
(11_env)learning_log$
```

Django applies the new migration, and the result is OK ❶.

We can verify that the migration worked as expected in the shell session, like this:

```
❶ >>> from learning_logs.models import Topic
❷ >>> for topic in Topic.objects.all():
...     print(topic, topic.owner)
...
Chess ll_admin
Rock Climbing ll_admin
>>>
```

We import `Topic` from `learning_logs.models` ❶, and then loop through all existing topics, printing each topic and the user it belongs to ❷. You can see that each topic now belongs to the user `ll_admin`. (If you get an error when you run this code, try exiting the shell and starting a new shell.)

NOTE

You can simply reset the database instead of migrating, but that will lose all existing data. It's good practice to learn how to migrate a database while maintaining the integrity of users' data. If you do want to start with a fresh database, issue the command `python manage.py flush` to rebuild the database structure. You'll have to create a new superuser, and all of your data will be gone.

Restricting Topics Access to Appropriate Users

Currently, if you're logged in, you'll be able to see all the topics, no matter which user you're logged in as. We'll change that by showing users only the topics that belong to them.

Make the following change to the `topics()` function in `views.py`:

```
views.py
--snip--
@login_required
def topics(request):
    """Show all topics."""
    topics = Topic.objects.filter(owner=request.user).order_by('date_added')
    context = {'topics': topics}
    return render(request, 'learning_logs/topics.html', context)
--snip--
```

When a user is logged in, the `request` object has a `request.user` attribute set that stores information about the user. The query `Topic.objects.filter(owner=request.user)` tells Django to retrieve only the `Topic` objects from the database whose `owner` attribute matches the current user. Because we're not changing how the topics are displayed, we don't need to change the template for the topics page at all.

To see if this works, log in as the user you connected all existing topics to, and go to the topics page. You should see all the topics. Now log out, and log back in as a different user. The topics page should list no topics.

Protecting a User's Topics

We haven't restricted access to the topic pages yet, so any registered user could try a bunch of URLs, like `http://localhost:8000/topics/1/`, and retrieve topic pages that happen to match.

Try it yourself. While logged in as the user that owns all topics, copy the URL or note the ID in the URL of a topic, and then log out and log back in as a different user. Enter that topic's URL. You should be able to read the entries, even though you're logged in as a different user.

We'll fix this now by performing a check before retrieving the requested entries in the `topic()` view function:

```
views.py    from django.shortcuts import render, redirect
              from django.contrib.auth.decorators import login_required
❶  from django.http import Http404

--snip--
@login_required
def topic(request, topic_id):
    """Show a single topic and all its entries."""
    topic = Topic.objects.get(id=topic_id)
    # Make sure the topic belongs to the current user.
❷  if topic.owner != request.user:
        raise Http404

    entries = topic.entry_set.order_by('-date_added')
    context = {'topic': topic, 'entries': entries}
    return render(request, 'learning_logs/topic.html', context)
--snip--
```

A 404 response is a standard error response that's returned when a requested resource doesn't exist on a server. Here we import the `Http404` exception ❶, which we'll raise if the user requests a topic they shouldn't see. After receiving a topic request, we make sure the topic's user matches the currently logged in user before rendering the page. If the current user doesn't own the requested topic, we raise the `Http404` exception ❷, and Django returns a 404 error page.

Now if you try to view another user's topic entries, you'll see a *Page Not Found* message from Django. In Chapter 20, we'll configure the project so users will see a proper error page.

Protecting the edit_entry Page

The `edit_entry` pages have URLs in the form `http://localhost:8000/edit_entry/entry_id/`, where the `entry_id` is a number. Let's protect this page so no one can use the URL to gain access to someone else's entries:

```
views.py    --snip--
@login_required
def edit_entry(request, entry_id):
    """Edit an existing entry."""
    entry = Entry.objects.get(id=entry_id)
```

```
topic = entry.topic
if topic.owner != request.user:
    raise Http404

if request.method != 'POST':
    --snip--
```

We retrieve the entry and the topic associated with this entry. We then check whether the owner of the topic matches the currently logged in user; if they don't match, we raise an `Http404` exception.

Associating New Topics with the Current User

Currently, our page for adding new topics is broken, because it doesn't associate new topics with any particular user. If you try adding a new topic, you'll see the error message `IntegrityError` along with `NOT NULL constraint failed: learning_logs_topic.owner_id`. Django's saying you can't create a new topic without specifying a value for the topic's `owner` field.

There's a straightforward fix for this problem, because we have access to the current user through the `request` object. Add the following code, which associates the new topic with the current user:

```
views.py
--snip--
@login_required
def new_topic(request):
    """Add a new topic."""
    if request.method != 'POST':
        # No data submitted; create a blank form.
        form = TopicForm()
    else:
        # POST data submitted; process data.
        form = TopicForm(data=request.POST)
        if form.is_valid():
           ❶           new_topic = form.save(commit=False)
           ❷           new_topic.owner = request.user
           ❸           new_topic.save()
            return redirect('learning_logs:topics')

    # Display a blank or invalid form.
    context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
--snip--
```

When we first call `form.save()`, we pass the `commit=False` argument because we need to modify the new topic before saving it to the database ❶. We then set the new topic's `owner` attribute to the current user ❷. Finally, we call `save()` on the topic instance just defined ❸. Now the topic has all the required data and will save successfully.

You should be able to add as many new topics as you want for as many different users as you want. Each user will have access only to their own data, whether they're viewing data, entering new data, or modifying old data.

TRY IT YOURSELF

19-3. Refactoring: There are two places in `views.py` where we make sure the user associated with a topic matches the currently logged in user. Put the code for this check in a function called `check_topic_owner()`, and call this function where appropriate.

19-4. Protecting new_entry: Currently, a user can add a new entry to another user's learning log by entering a URL with the ID of a topic belonging to another user. Prevent this attack by checking that the current user owns the entry's topic before saving the new entry.

19-5. Protected Blog: In your Blog project, make sure each blog post is connected to a particular user. Make sure all posts are publicly accessible but only registered users can add posts and edit existing posts. In the view that allows users to edit their posts, make sure the user is editing their own post before processing the form.

Summary

In this chapter, you learned to use forms to allow users to add new topics and entries, and edit existing entries. You then learned how to implement user accounts. You allowed existing users to log in and out, and used Django's default `UserCreationForm` to let people create new accounts.

After building a simple user authentication and registration system, you restricted access to logged-in users for certain pages using the `@login_required` decorator. You then attributed data to specific users through a foreign key relationship. You also learned to migrate the database when the migration requires you to specify some default data.

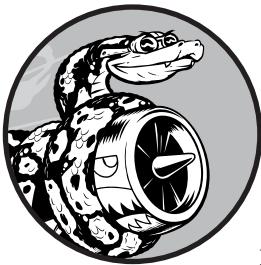
Finally, you learned how to make sure a user can only see data that belongs to them by modifying the view functions. You retrieved appropriate data using the `filter()` method and compared the owner of the requested data to the currently logged in user.

It might not always be immediately obvious what data you should make available and what data you should protect, but this skill will come with practice. The decisions we've made in this chapter to secure our users' data also illustrate why working with others is a good idea when building a project: having someone else look over your project makes it more likely that you'll spot vulnerable areas.

You now have a fully functioning project running on your local machine. In the final chapter, you'll style Learning Log to make it visually appealing, and you'll deploy the project to a server so anyone with internet access can register and make an account.

20

STYLING AND DEPLOYING AN APP



Learning Log is fully functional now, but it has no styling and runs only on your local machine. In this chapter, you'll style the project in a simple but professional manner and then deploy it to a live server so anyone in the world can make an account and use it.

For the styling we'll use the Bootstrap library, a collection of tools for styling web applications so they look professional on all modern devices, from a large flat-screen monitor to a smartphone. To do this, we'll use the `django-bootstrap4` app, which will also give you practice using apps made by other Django developers.

We'll deploy Learning Log using Heroku, a site that lets you push your project to one of its servers, making it available to anyone with an internet connection. We'll also start using a version control system called Git to track changes to the project.

When you're finished with Learning Log, you'll be able to develop simple web applications, make them look good, and deploy them to a live server. You'll also be able to use more advanced learning resources as you develop your skills.

Styling Learning Log

We've purposely ignored styling until now to focus on Learning Log's functionality first. This is a good way to approach development, because an app is useful only if it works. Of course, once it's working, appearance is critical so people will want to use it.

In this section, I'll introduce the django-bootstrap4 app and show you how to integrate it into a project to make it ready for live deployment.

The `django-bootstrap4` App

We'll use django-bootstrap4 to integrate Bootstrap into our project. This app downloads the required Bootstrap files, places them in an appropriate location in your project, and makes the styling directives available in your project's templates.

To install django-bootstrap4, issue the following command in an active virtual environment:

```
(11_env)learning_log$ pip install django-bootstrap4  
--snip--  
Successfully installed django-bootstrap4-0.0.7
```

Next, we need to add the following code to include django-bootstrap4 in `INSTALLED_APPS` in `settings.py`:

```
settings.py  
--snip--  
INSTALLED_APPS = [  
    # My apps.  
    'learning_logs',  
    'users',  
  
    # Third party apps.  
    'bootstrap4',  
  
    # Default django apps.  
    'django.contrib.admin',  
    --snip--
```

Start a new section called *Third party apps* for apps created by other developers and add '`bootstrap4`' to this section. Make sure you place this section after `# My apps` but before the section containing Django's default apps.

Using Bootstrap to Style Learning Log

Bootstrap is a large collection of styling tools. It also has a number of templates you can apply to your project to create an overall style. It's much easier to use these templates than it is to use individual styling tools. To see the templates Bootstrap offers, go to <https://getbootstrap.com/>, click **Examples**, and look for the *Navbars* section. We'll use the *Navbar static* template, which provides a simple top navigation bar and a container for the page's content.

Figure 20-1 shows what the home page will look like after we apply Bootstrap's template to *base.html* and modify *index.html* slightly.

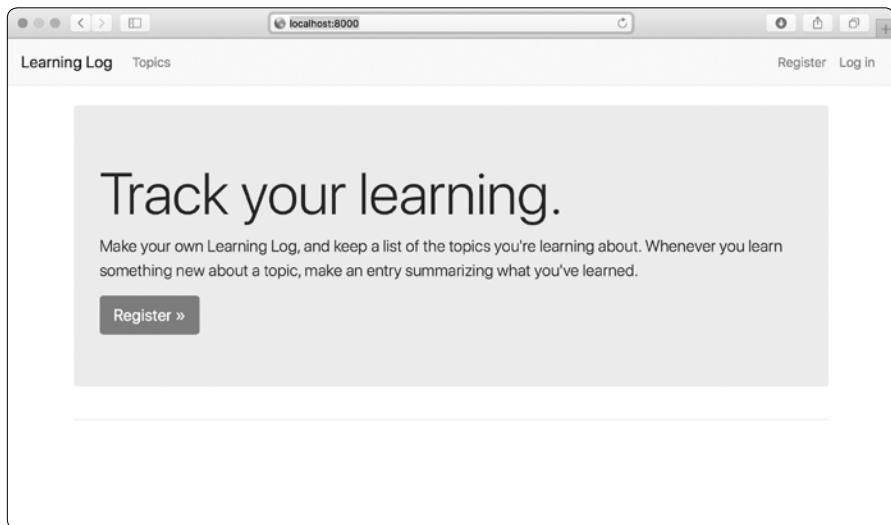


Figure 20-1: The Learning Log home page using Bootstrap

Modifying *base.html*

We need to modify the *base.html* template to accommodate the Bootstrap template. I'll introduce the new *base.html* in parts.

Defining the HTML Headers

The first change we'll make to *base.html* defines the HTML headers in the file, so whenever a Learning Log page is open, the browser title bar displays the site name. We'll also add some requirements for using Bootstrap in our templates. Delete everything in *base.html* and replace it with the following code:

```
base.html ❶ {%- load bootstrap4 %}

❷ <!doctype html>
❸ <html lang="en">
❹ <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,
        shrink-to-fit=no">
❺ <title>Learning Log</title>

❻ {%- bootstrap_css %}
    {%- bootstrap_javascript jquery='full' %}

❼ </head>
```

At ❶ we load the collection of template tags available in django-bootstrap4. Next, we declare this file as an HTML document ❷ written in English ❸. An HTML file is divided into two main parts, the *head* and the *body*—the head of the file begins at ❹. The head of an HTML file doesn’t contain any content: it just tells the browser what it needs to know to display the page correctly. At ❺ we include a title element for the page, which will display in the browser’s title bar whenever Learning Log is open.

At ❻ we use one of django-bootstrap4’s custom template tags, which tells Django to include all the Bootstrap style files. The tag that follows enables all the interactive behavior you might use on a page, such as collapsible navigation bars. At ❼ is the closing `</head>` tag.

Defining the Navigation Bar

The code that defines the navigation bar at the top of the page is fairly long, because it has to work well on narrow phone screens and wide desktop monitors. We’ll work through the navigation bar in sections.

Here’s the first part of the navigation bar:

```
base.html
--snip--
</head>
❶ <body>

❷   <nav class="navbar navbar-expand-md navbar-light bg-light mb-4 border">

❸     <a class="navbar-brand" href="{% url 'learning_logs:index'%}">
          Learning Log</a>

❹     <button class="navbar-toggler" type="button" data-toggle="collapse"
            data-target="#navbarCollapse" aria-controls="navbarCollapse"
            aria-expanded="false" aria-label="Toggle navigation">
           <span class="navbar-toggler-icon"></span></button>
```

The first element is the opening `<body>` tag ❶. The *body* of an HTML file contains the content users will see on a page. At ❷ is a `<nav>` element that indicates the page’s navigation links section. Everything contained in this element is styled according to the Bootstrap style rules defined by the selectors `navbar`, `navbar-expand-md`, and the rest that you see here. A *selector* determines which elements on a page a certain style rule applies to. The `navbar-light` and `bg-light` selectors style the navigation bar with a light-themed background. The `mb` in `mb-4` is short for `margin-bottom`; this selector ensures that a little space appears between the navigation bar and the rest of the page. The `border` selector provides a thin border around the light background to set it off a little from the rest of the page.

At ❸ we set the project’s name to appear at the far left of the navigation bar and make it a link to the home page; it will appear on every page in the project. The `navbar-brand` selector styles this link so it stands out from the rest of the links and is a way of branding the site.

At ❹ the template defines a button that appears if the browser window is too narrow to display the whole navigation bar horizontally. When the

user clicks the button, the navigation elements will appear in a drop-down list. The `collapse` reference causes the navigation bar to collapse when the user shrinks the browser window or when the site is displayed on mobile devices with small screens.

Here's the next section of code that defines the navigation bar:

```
base.html
        --snip--
        <span class="navbar-toggler-icon"></span></button>
❶    <div class="collapse navbar-collapse" id="navbarCollapse">
❷        <ul class="navbar-nav mr-auto">
❸            <li class="nav-item">
                <a class="nav-link" href="{% url 'learning_logs:topics'%}">
                    Topics</a></li>
            </ul>
```

At ❶ we open a new section of the navigation bar. The term `div` is short for division; you build a web page by dividing it into sections and defining style and behavior rules that apply to that section. Any styling or behavior rules that are defined in an opening `div` tag affect everything you see until the next closing `div` tag, which is written as `</div>`. This is the beginning of the part of the navigation bar that will be collapsed on narrow screens and windows.

At ❷ we define a new set of links. Bootstrap defines navigation elements as items in an unordered list with style rules that make it look nothing like a list. Every link or element you need on the bar can be included as an item in one of these lists. Here, the only item in the list is our link to the Topics page ❸.

Here's the next part of the navigation bar:

```
base.html
        --snip--
        </ul>
❶    <ul class="navbar-nav ml-auto">
❷        {% if user.is_authenticated %}
            <li class="nav-item">
                <span class="navbar-text">Hello, {{ user.username }}.</span>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'users:logout' %}">Log out</a>
            </li>
        {% else %}
            <li class="nav-item">
                <a class="nav-link" href="{% url 'users:register' %}">Register</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'users:login' %}">Log in</a></li>
        {% endif %}
        </ul>
❸    </div>

    </nav>
```

At ❶ we begin a new set of links by using another opening `` tag. You can have as many groups of links as you need on a page. This will be the group of links related to login and registration that appears on the right side of the navigation bar. The selector `ml-auto` is short for *margin-left-automatic*: this selector examines the other elements in the navigation bar and works out a left margin that pushes this group of links to the right side of the screen.

The `if` block at ❷ is the same conditional block we used earlier to display appropriate messages to users depending on whether or not they're logged in. The block is a little longer now because some styling rules are inside the conditional tags. At ❸ is a `` element. The *span element* styles pieces of text, or elements of a page, that are part of a longer line. Whereas `div` elements create their own division in a page, `span` elements are continuous within a larger section. This can be confusing at first, because many pages have deeply nested `div` elements. Here, we're using the `span` element to style informational text on the navigation bar, such as the logged-in user's name. We want this information to appear different from a link, so users aren't tempted to click these elements.

At ❹ we close the `div` element that contains the parts of the navigation bar that will collapse on narrow screens, and at the end of this section we close the navigation bar overall. If you wanted to add more links to the navigation bar, you'd add another `` item to any of the `` groups that we've defined in the navigation bar by using identical styling directives as what you've seen here.

There's still a bit more we need to add to `base.html`. We need to define two blocks that the individual pages can use to place the content specific to those pages.

Defining the Main Part of the Page

The rest of `base.html` contains the main part of the page:

```
base.html
        --snip--
    </nav>

❶  <main role="main" class="container">
❷    <div class="pb-2 mb-2 border-bottom">
        {% block page_header %}{% endblock page_header %}
    </div>
❸    <div>
        {% block content %}{% endblock content %}
    </div>
</main>

</body>

</html>
```

At ❶ we open a `<main>` tag. The *main* element is used for the most significant part of the body of a page. Here we assign the bootstrap selector

container, which is a simple way to group elements on a page. We'll place two div elements in this container.

The first div element ❷ contains a `page_header` block. We'll use this block to title most pages. To make this section stand out from the rest of the page, we place some padding below the header. *Padding* refers to space between an element's content and its border. The selector `pb-2` is a bootstrap directive that provides a moderate amount of padding at the bottom of the styled element. A *margin* is the space between an element's border and other elements on the page. We want a border only on the bottom of the page, so we use the selector `border-bottom`, which provides a thin border at the bottom of the `page_header` block.

At ❸ we define one more div element, which contains the block content. We don't apply any specific style to this block, so we can style the content of any page as we see fit for that page. We end the `base.html` file with closing tags for the `main`, `body`, and `html` elements.

When you load Learning Log's home page in a browser, you should see a professional-looking navigation bar that matches the one shown in Figure 20-1. Try resizing the window so it's very narrow; a button should replace the navigation bar. Click the button, and all the links should appear in a drop-down list.

Styling the Home Page Using a Jumbotron

To update the home page, we'll use a Bootstrap element called a *jumbotron*, which is a large box that stands out from the rest of the page and can contain anything you want. Typically, it's used on home pages to hold a brief description of the overall project and a call to action that invites the viewer to get involved.

Here's the revised `index.html` file:

```
index.html  {% extends "learning_logs/base.html" %}

❶  {% block page_header %}
❷    <div class="jumbotron">
❸      <h1 class="display-3">Track your learning.</h1>

❹      <p class="lead">Make your own Learning Log, and keep a list of the
          topics you're learning about. Whenever you learn something new
          about a topic, make an entry summarizing what you've learned.</p>

❺      <a class="btn btn-lg btn-primary" href="{% url 'users:register' %}"
          role="button">Register &raquo;</a>
    </div>
❻  {% endblock page_header %}
```

At ❶ we tell Django that we're about to define what goes in the `page_header` block. A jumbotron is just a div element with a set of styling directives applied to it ❷. The `jumbotron` selector applies this group of styling directives from the Bootstrap library to this element.

Inside the jumbotron are three elements. The first is a short message, *Track your learning*, that gives first-time visitors a sense of what Learning Log does. The `h1` class is a first-level header, and the `display-3` selector adds a thinner and taller look to this particular header ❸. At ❹ we include a longer message that provides more information about what the user can do with their learning log.

Rather than just using a text link, we create a button at ❺ that invites users to register their Learning Log account. This is the same link as in the header, but the button stands out on the page and shows the viewer what they need to do to start using the project. The selectors you see here style this as a large button that represents a call to action. The code `»` is an *HTML entity* that looks like two right angle brackets combined (`>>`). At ❻ we close the `page_header` block. We aren't adding any more content to this page, so we don't need to define the `content` block on this page.

The index page now looks like Figure 20-1 and is a significant improvement over our unstyled project.

Styling the Login Page

We've refined the overall appearance of the login page but not the login form yet. Let's make the form look consistent with the rest of the page by modifying the `login.html` file:

```
login.html  {% extends "learning_logs/base.html" %}  
❶  {% load bootstrap4 %}  
  
❷  {% block page_header %}  
      <h2>Log in to your account.</h2>  
  {% endblock page_header %}  
  
  {% block content %}  
❸    <form method="post" action="{% url 'users:login' %}" class="form">  
        {% csrf_token %}  
❹        {% bootstrap_form form %}  
❺        {% buttons %}  
            <button name="submit" class="btn btn-primary">Log in</button>  
        {% endbuttons %}  
  
        <input type="hidden" name="next"  
              value="{% url 'learning_logs:index' %}" />  
    </form>  
  
  {% endblock content %}
```

At ❶ we load the `bootstrap4` template tags into this template. At ❷ we define the `page_header` block, which tells the user what the page is for. Notice that we've removed the `{% if form.errors %}` block from the template; `django-bootstrap4` manages form errors automatically.

At ❸ we add a `class="form"` attribute, and then we use the template tag `{% bootstrap_form %}` when we display the form ❹; this replaces the `{{ form.as_p }}` tag we were using in Chapter 19. The `{% bootstrap_form %}` template

tag inserts Bootstrap style rules into the form's individual elements as the form is rendered. At ❸ we open a bootstrap4 template tag `{% buttons %}`, which adds Bootstrap styling to buttons.

Figure 20-2 shows the login form now. The page is much cleaner and has consistent styling and a clear purpose. Try logging in with an incorrect username or password; you'll see that even the error messages are styled consistently and integrate well with the overall site.

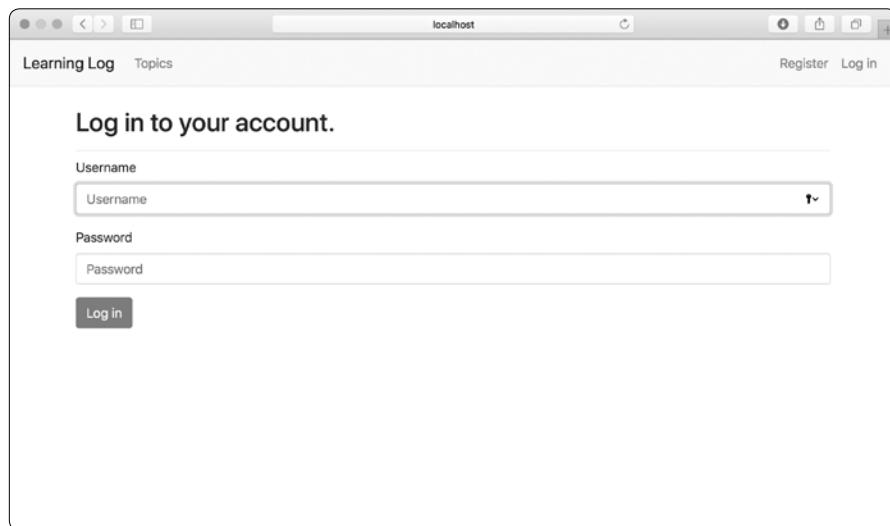


Figure 20-2: The login page styled with Bootstrap

Styling the Topics Page

Let's make sure the pages for viewing information are styled appropriately as well, starting with the topics page:

```
topics.html
{% extends "learning_logs/base.html" %}

❶ {% block page_header %}
    <h1>Topics</h1>
    {% endblock page_header %}

    {% block content %}
        <ul>
            {% for topic in topics %}
                ❷   <li><h3>
                    <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
                    </h3></li>
                {% empty %}
                    <li><h3>No topics have been added yet.</h3></li>
                {% endfor %}
            </ul>

❸   <h3><a href="{% url 'learning_logs:new_topic' %}">Add a new topic</a></h3>
    {% endblock content %}
```

We don't need the `{% load bootstrap4 %}` tag, because we're not using any custom bootstrap4 template tags in this file. We move the heading *Topics* into the `page_header` block and give it a header styling instead of using the simple paragraph tag ❶. We style each topic as an `<h3>` element to make them a little larger on the page ❷ and do the same for the link to add a new topic ❸.

Styling the Entries on the Topic Page

The topic page has more content than most pages, so it needs a bit more work. We'll use Bootstrap's card component to make each entry stand out. A *card* is a div with a set of flexible, predefined styles that's perfect for displaying a topic's entries:

topic.html

```
{% extends 'learning_logs/base.html' %}

❶ {% block page_header %}
    <h3>{{ topic }}</h3>
{% endblock page_header %}

{% block content %}
<p>
    <a href="{% url 'learning_logs:new_entry' topic.id %}">Add new entry</a>
</p>

❷ {% for entry in entries %}
❸     <div class="card mb-3">
❹         <h4 class="card-header">
            {{ entry.date_added|date:'M d, Y H:i' }}
❺         <small><a href="{% url 'learning_logs:edit_entry' entry.id %}">
                edit entry</a></small>
        </h4>
❻         <div class="card-body">
            {{ entry.text|linebreaks }}
        </div>
    </div>
❾ {% empty %}
    <p>There are no entries for this topic yet.</p>
{% endfor %}

{% endblock content %}
```

We first place the topic in the `page_header` block ❶. Then we delete the unordered list structure previously used in this template. Instead of making each entry a list item, we create a div element with the selector `card` at ❷. This card has two nested elements: one to hold the timestamp and the link to edit the entry, and another to hold the body of the entry.

The first element in the card is a header, which is an `<h4>` element with the selector `card-header` ❸. This card header contains the date the entry was made and a link to edit the entry. The `<small>` tag around the `edit_entry` link

makes it appear a little smaller than the timestamp ❸. The second element is a div with the selector card-body ❹, which places the text of the entry in a simple box on the card. Notice that the Django code for including the information on the page hasn't changed; only the elements that affect the appearance of the page have changed.

Figure 20-3 shows the topic page with its new look. Learning Log's functionality hasn't changed, but it looks more professional and inviting to users now.

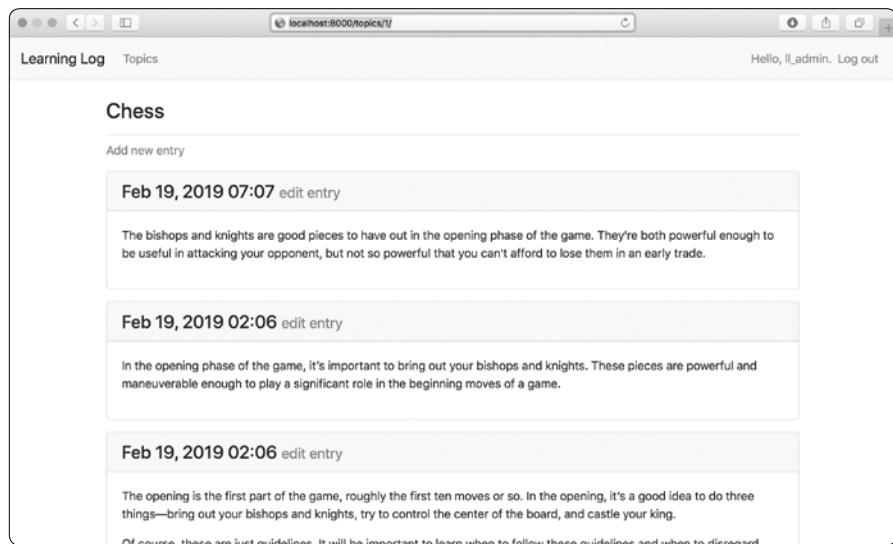


Figure 20-3: The topic page with Bootstrap styling

NOTE

If you want to use a different Bootstrap template, follow a similar process to what we've done so far in this chapter. Copy the template you want to use into base.html, and modify the elements that contain actual content so the template displays your project's information. Then use Bootstrap's individual styling tools to style the content on each page.

TRY IT YOURSELF

20-1. Other Forms: We applied Bootstrap's styles to the login page. Make similar changes to the rest of the form-based pages including new_topic, new_entry, edit_entry, and register.

20-2. Stylish Blog: Use Bootstrap to style the Blog project you created in Chapter 19.

Deploying Learning Log

Now that we have a professional-looking project, let's deploy it to a live server so anyone with an internet connection can use it. We'll use Heroku, a web-based platform that allows you to manage the deployment of web applications. We'll get Learning Log up and running on Heroku.

Making a Heroku Account

To make an account, go to <https://heroku.com/> and click one of the signup links. It's free to make an account, and Heroku has a free tier that allows you to test your projects in live deployment before properly deploying them.

NOTE

Heroku's free tier has limits, such as the number of apps you can deploy and how often people can visit your app. But these limits are generous enough to let you practice deploying apps without any cost.

Installing the Heroku CLI

To deploy and manage a project on Heroku's servers, you'll need the tools available in the Heroku Command Line Interface (CLI). To install the latest version of the Heroku CLI, visit <https://devcenter.heroku.com/articles/heroku-cli/> and follow the instructions for your operating system. The instructions will include either a one-line terminal command or an installer you can download and run.

Installing Required Packages

You'll also need to install three packages that help serve Django projects on a live server. In an active virtual environment, issue the following commands:

```
(11_env)learning_log$ pip install psycopg2==2.7.*  
(11_env)learning_log$ pip install django-heroku  
(11_env)learning_log$ pip install gunicorn
```

The `psycopg2` package is required to manage the database that Heroku uses. The `django-heroku` package handles almost the entire configuration our app needs to run properly on Heroku servers. This includes managing the database and storing static files in a place where they can be served properly. *Static files* contain style rules and JavaScript files. The `gunicorn` package provides a server capable of serving apps in a live environment.

Creating a requirements.txt File

Heroku needs to know which packages our project depends on, so we'll use `pip` to generate a file listing them. Again, from an active virtual environment, issue the following command:

```
(11_env)learning_log$ pip freeze > requirements.txt
```

The `freeze` command tells pip to write the names of all the packages currently installed in the project into the file `requirements.txt`. Open this file to see the packages and version numbers installed in your project:

`requirements.txt`

```
dj-database-url==0.5.0
Django==2.2.0
django-bootstrap4==0.0.7
django-heroku==0.3.1
gunicorn==19.9.0
psycopg2==2.7.7
pytz==2018.9
sqlparse==0.2.4
whitenoise==4.1.2
```

Learning Log already depends on eight different packages with specific version numbers, so it requires a specific environment to run properly. (We installed four of these packages manually, and four of them were installed automatically as dependencies of these packages.)

When we deploy Learning Log, Heroku will install all the packages listed in `requirements.txt`, creating an environment with the same packages we're using locally. For this reason, we can be confident the deployed project will behave the same as it does on our local system. This is a huge advantage as you start to build and maintain various projects on your system.

NOTE *If a package is listed on your system but the version number differs from what's shown here, keep the version you have on your system.*

Specifying the Python Runtime

Unless you specify a Python version, Heroku will use its current default version of Python. Let's make sure Heroku uses the same version of Python we're using. In an active virtual environment, issue the command `python --version`:

```
(11_env)learning_log$ python --version
Python 3.7.2
```

In this example I'm running Python 3.7.2. Make a new file called `runtime.txt` in the same directory as `manage.py`, and enter the following:

`runtime.txt`

```
python-3.7.2
```

This file should contain one line with your Python version specified in the format shown; make sure you enter `python` in lowercase, followed by a hyphen, followed by the three-part version number.

NOTE *If you get an error reporting that the Python runtime you requested isn't available, go to <https://devcenter.heroku.com/categories/language-support/> and look for a link to Specifying a Python Runtime. Scan through the article to find the available runtimes, and use the one that most closely matches your Python version.*

Modifying *settings.py* for Heroku

Now we need to add a section at the end of *settings.py* to define some specific settings for the Heroku environment:

```
settings.py
--snip--
# My settings
LOGIN_URL = 'users:login'

# Heroku settings.
import django_heroku
django_heroku.settings(locals())
```

Here we import the `django_heroku` module and call the `settings()` function. This function modifies some settings that need specific values for the Heroku environment.

Making a *Procfile* to Start Processes

A *Procfile* tells Heroku which processes to start to properly serve the project. Save this one-line file as *Procfile*, with an uppercase *P* and no file extension, in the same directory as *manage.py*.

Here's the line that goes in *Procfile*:

```
Procfile    web: gunicorn learning_log.wsgi --log-file -
```

This line tells Heroku to use gunicorn as a server and to use the settings in *learning_log/wsgi.py* to launch the app. The `log-file` flag tells Heroku the kinds of events to log.

Using Git to Track the Project's Files

As discussed in Chapter 17, Git is a version control program that allows you to take a snapshot of the code in your project each time you implement a new feature successfully. If anything goes wrong, you can easily return to the last working snapshot of your project; for example, if you accidentally introduce a bug while working on a new feature. Each snapshot is called a *commit*.

Using Git, you can try implementing new features without worrying about breaking your project. When you're deploying to a live server, you need to make sure you're deploying a working version of your project. To read more about Git and version control, see Appendix D.

Installing Git

Git may already be installed on your system. To find out if Git is already installed, open a new terminal window and issue the command `git --version`:

```
(11_env)learning_log$ git --version
git version 2.17.0
```

If you get an error message for some reason, see the installation instructions for Git in Appendix D.

Configuring Git

Git keeps track of who makes changes to a project, even when only one person is working on the project. To do this, Git needs to know your username and email. You must provide your username, but you can make up an email for your practice projects:

```
(11_env)learning_log$ git config --global user.name "ehmatthes"  
(11_env)learning_log$ git config --global user.email "eric@example.com"
```

If you forget this step, Git will prompt you for this information when you make your first commit.

Ignoring Files

We don't need Git to track every file in the project, so we'll tell it to ignore some files. Create a file called `.gitignore` in the folder that contains `manage.py`. Notice that this filename begins with a dot and has no file extension. Here's the code that goes in `.gitignore`:

```
.gitignore  
11_env/  
__pycache__/  
*.sqlite3
```

We tell Git to ignore the entire `11_env` directory, because we can re-create it automatically at any time. We also don't track the `__pycache__` directory, which contains the `.pyc` files that are created automatically when Django runs the `.py` files. We don't track changes to the local database, because it's a bad habit: if you're ever using SQLite on a server, you might accidentally overwrite the live database with your local test database when you push the project to the server. The asterisk in `*.sqlite3` tells Git to ignore any file that ends with the extension `.sqlite3`.

NOTE

If you're using macOS, add `.DS_Store` to your `.gitignore` file. This is a file that stores information about folder settings on macOS, and it has nothing to do with this project.

Making Hidden Files Visible

Most operating systems hide files and folders that begin with a dot, such as `.gitignore`. When you open a file browser or try to open a file from an application such as Sublime Text, you won't see these kinds of files by default. But as a programmer, you'll need to see them. Here's how to view hidden files, depending on your operating system:

- On Windows, open Windows Explorer, and then open a folder such as `Desktop`. Click the **View** tab, and make sure **File name extensions** and **Hidden items** are checked.

- On macOS, you can press ⌘-SHIFT-. (dot) in any file browser window to see hidden files and folders.
- On Linux systems such as Ubuntu, you can press CTRL-H in any file browser to display hidden files and folders. To make this setting permanent, open a file browser such as Nautilus and click the options tab (indicated by three lines). Select the **Show Hidden Files** checkbox.

Committing the Project

We need to initialize a Git repository for Learning Log, add all the necessary files to the repository, and commit the initial state of the project. Here's how to do that:

```
❶ (11_env)learning_log$ git init
Initialized empty Git repository in /home/ehmatthes/pcc/learning_log/.git/
❷ (11_env)learning_log$ git add .
❸ (11_env)learning_log$ git commit -am "Ready for deployment to heroku."
[master (root-commit) 79fef72] Ready for deployment to heroku.
 45 files changed, 712 insertions(+)
  create mode 100644 .gitignore
  create mode 100644 Procfile
  --snip--
  create mode 100644 users/views.py
❹ (11_env)learning_log$ git status
On branch master
nothing to commit, working tree clean
(11_env)learning_log$
```

At ❶ we issue the `git init` command to initialize an empty repository in the directory containing Learning Log. At ❷ we use the `git add .` command, which adds all the files that aren't being ignored to the repository. (Don't forget the dot.) At ❸ we issue the command `git commit -am commit message`: the `-a` flag tells Git to include all changed files in this commit, and the `-m` flag tells Git to record a log message.

Issuing the `git status` command ❹ indicates that we're on the *master* branch and that our working tree is *clean*. This is the status you'll want to see any time you push your project to Heroku.

Pushing to Heroku

We're finally ready to push the project to Heroku. In an active virtual environment, issue the following commands:

```
❶ (11_env)learning_log$ heroku login
heroku: Press any key to open up the browser to login or q to exit:
Logging in... done
Logged in as eric@example.com
❷ (11_env)learning_log$ heroku create
Creating app... done, ⬤ secret-lowlands-82594
https://secret-lowlands-82594.herokuapp.com/ |
https://git.heroku.com/secret-lowlands-82594.git
```

```
❸ (ll_env)learning_log$ git push heroku master
--snip--
remote: -----> Launching...
remote:           Released v5
❹ remote:           https://secret-lowlands-82594.herokuapp.com/ deployed to Heroku
remote: Verifying deploy... done.
To https://git.heroku.com/secret-lowlands-82594.git
 * [new branch]      master -> master
(ll_env)learning_log$
```

First you issue the `heroku login` command, which will take you to a page in your browser where you can log in to your Heroku account ❶. Then you tell Heroku to build an empty project ❷. Heroku generates a name made up of two words and a number; you can change this later on. Next, we issue the command `git push heroku master` ❸, which tells Git to push the master branch of the project to the repository Heroku just created. Then Heroku builds the project on its servers using these files. At ❹ is the URL we'll use to access the live project, which we can change along with the project name.

When you've issued these commands, the project is deployed but not fully configured. To check that the server process started correctly, use the `heroku ps` command:

```
(ll_env)learning_log$ heroku ps
❶ Free dyno hours quota remaining this month: 450h 44m (81%)
Free dyno usage for this app: 0h 0m (0%)
For more information on dyno sleeping and how to upgrade, see:
https://devcenter.heroku.com/articles/dyno-sleeping
❷ === web (Free): gunicorn learning_log.wsgi --log-file - (1)
web.1: up 2019/02/19 23:40:12 -0900 (~ 10m ago)
(ll_env)learning_log$
```

The output shows how much more time the project can be active in the next month ❶. At the time of this writing, Heroku allows free deployments to be active for up to 550 hours in a month. If a project exceeds this limit, a standard server error page will display; we'll customize this error page shortly. At ❷ we see that the process defined in *Procfile* has been started.

Now we can open the app in a browser using the command `heroku open`:

```
(ll_env)learning_log$ heroku open
(ll_env)learning_log$
```

This command spares you from opening a browser and entering the URL Heroku showed you, but that's another way to open the site. You should see the home page for Learning Log, styled correctly. However, you can't use the app yet because we haven't set up the database.

NOTE

Heroku's deployment process changes from time to time. If you have any unresolvable issues, look at Heroku's documentation for help. Go to <https://devcenter.heroku.com/>, click Python, and look for a link to Get Started with Python or Deploying Python and Django Apps on Heroku. If you don't understand what you see there, check out the suggestions in Appendix C.

Setting Up the Database on Heroku

We need to run `migrate` once to set up the live database and apply all the migrations we generated during development. You can run Django and Python commands on a Heroku project using the command `heroku run`. Here's how to run `migrate` on the Heroku deployment:

```
❶ (11_env)learning_log$ heroku run python manage.py migrate
❷ Running 'python manage.py migrate' on ⬤ secret-lowlands-82594... up, run.3060
    --snip--
❸ Running migrations:
    --snip--
        Applying learning_logs.0001_initial... OK
        Applying learning_logs.0002_entry... OK
        Applying learning_logs.0003_topic_owner... OK
        Applying sessions.0001_initial... OK
(11_env)learning_log$
```

We first issue the command `heroku run python manage.py migrate` **❶**. Heroku then creates a terminal session to run the `migrate` command **❷**. At **❸** Django applies the default migrations and the migrations we generated during the development of Learning Log.

Now when you visit your deployed app, you should be able to use it just as you did on your local system. But you won't see any of the data you entered on your local deployment, including your superuser account, because we didn't copy the data to the live server. This is normal practice: you don't usually copy local data to a live deployment because the local data is usually test data.

You can share your Heroku link to let anyone use your version of Learning Log. In the next section, we'll complete a few more tasks to finish the deployment process and set you up to continue developing Learning Log.

Refining the Heroku Deployment

Now we'll refine the deployment by creating a superuser, just as we did locally. We'll also make the project more secure by changing the setting `DEBUG` to `False`, so users won't see any extra information in error messages that they could use to attack the server.

Creating a Superuser on Heroku

You've already seen that we can run one-off commands using the `heroku run` command. But you can also run commands by opening a Bash terminal session while connected to the Heroku server using the command `heroku run bash`. Bash is the language that runs in many Linux terminals. We'll use the Bash terminal session to create a superuser so we can access the admin site on the live app:

```
(11_env)learning_log$ heroku run bash
Running 'bash' on ⬤ secret-lowlands-82594... up, run.9858
❶ ~ $ ls
learning_log learning_logs manage.py Procfile requirements.txt runtime.txt
```

```
staticfiles users
❷ ~ $ python manage.py createsuperuser
Username (leave blank to use 'u47318'): ll_admin
Email address:
Password:
Password (again):
Superuser created successfully.
❸ ~ $ exit
exit
(ll_env)learning_log$
```

At ❶ we run `ls` to see which files and directories exist on the server, which should be the same files we have on our local system. You can navigate this filesystem like any other.

NOTE

Windows users will use the same commands shown here (such as `ls` instead of `dir`), because you're running a Linux terminal through a remote connection.

At ❷ we run the command to create a superuser, which outputs the same prompts we saw on our local system when we created a superuser in Chapter 18. When you're finished creating the superuser in this terminal session, run the `exit` command to return to your local system's terminal session ❸.

Now you can add `/admin/` to the end of the URL for the live app and log in to the admin site. For me, the URL is `https://secret-lowlands-82594.herokuapp.com/admin/`.

If others have already started using your project, be aware that you'll have access to all of their data! Don't take this lightly, and users will continue to trust you with their data.

Creating a User-Friendly URL on Heroku

Most likely, you'll want your URL to be friendlier and more memorable than `https://secret-lowlands-82594.herokuapp.com/`. You can rename the app using a single command:

```
(ll_env)learning_log$ heroku apps:rename learning-log
Renaming secret-lowlands-82594 to learning-log-2e... done
https://learning-log.herokuapp.com/ | https://git.heroku.com/learning-log.git
Git remote heroku updated
❸ Don't forget to update git remotes for all other local checkouts of the app.
(ll_env)learning_log$
```

You can use letters, numbers, and dashes when naming your app, and call it whatever you want, as long as no one else has claimed the name. This deployment now lives at `https://learning-log.herokuapp.com/`. The project is no longer available at the previous URL; the `apps:rename` command completely moves the project to the new URL.

NOTE

When you deploy your project using Heroku's free service, Heroku puts your deployment to sleep if it hasn't received any requests after a certain amount of time or if it's been too active for the free tier. The first time a user accesses the site after it's been sleeping, it will take longer to load, but the server will respond to subsequent requests more quickly. This is how Heroku can afford to offer free deployments.

Securing the Live Project

One glaring security issue exists in the way our project is currently deployed: the setting DEBUG=True in *settings.py*, which provides debug messages when errors occur. Django's error pages give you vital debugging information when you're developing a project; however, they give way too much information to attackers if you leave them enabled on a live server.

We'll control whether debugging information is shown on the live site by setting an environment variable. *Environment variables* are values set in a specific environment. This is one of the ways sensitive information is stored on a server, keeping it separate from the rest of the project's code.

Let's modify *settings.py* so it looks for an environment variable when the project is running on Heroku:

```
settings.py
--snip--
# Heroku settings.
import django_heroku
django_heroku.settings(locals())

if os.environ.get('DEBUG') == 'TRUE':
    DEBUG = True
elif os.environ.get('DEBUG') == 'FALSE':
    DEBUG = False
```

The method `os.environ.get()` reads the value associated with a specific environment variable in any environment where the project is running. If the variable we're asking for is set, the method returns its value; if it's not set, the method returns `None`. Using environment variables to store Boolean values can be confusing. In most cases, environment variables are stored as strings, and you have to be careful about this. Consider this snippet from a simple Python terminal session:

```
>>> bool('False')
True
```

The Boolean value of the string '`False`' is `True`, because any non-empty string evaluates to `True`. So we'll use the strings '`TRUE`' and '`FALSE`', in all capitals, to be clear that we're not storing Python's actual `True` and `False` Boolean values. When Django reads in the environment variable with the key '`DEBUG`' on Heroku, we'll set `DEBUG` to `True` if the value is '`TRUE`' and `False` if the value is '`FALSE`'.

Committing and Pushing Changes

Now we need to commit the changes made to `settings.py` to the Git repository, and then push the changes to Heroku. Here's a terminal session showing this process:

```
❶ (ll_env)learning_log$ git commit -am "Set DEBUG based on environment variables."
[master 3427244] Set DEBUG based on environment variables.
 1 file changed, 4 insertions(+)
❷ (ll_env)learning_log$ git status
On branch master
nothing to commit, working tree clean
(ll_env)learning_log$
```

We issue the `git commit` command with a short but descriptive commit message ❶. Remember that the `-am` flag makes sure Git commits all the files that have changed and records the log message. Git recognizes that one file has changed and commits this change to the repository.

At ❷ the status shows that we're working on the master branch of the repository and that there are now no new changes to commit. It's essential that you check the status for this message before pushing to Heroku. If you don't see this message, some changes haven't been committed, and those changes won't be pushed to the server. You can try issuing the `commit` command again, but if you're not sure how to resolve the issue, read through Appendix D to better understand how to work with Git.

Now let's push the updated repository to Heroku:

```
(ll_env)learning_log$ git push heroku master
remote: Building source:
remote:
remote: -----> Python app detected
remote: -----> Installing requirements with pip
--snip--
remote: -----> Launching...
remote: Released v6
remote: https://learning-log.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/learning-log.git
 144f020..d5075a1  master -> master
(ll_env)learning_log$
```

Heroku recognizes that the repository has been updated, and it rebuilds the project to make sure all the changes have been taken into account. It doesn't rebuild the database, so we won't have to run `migrate` for this update.

Setting Environment Variables on Heroku

Now we can set the value we want for DEBUG in *settings.py* through Heroku. The command `heroku config:set` sets an environment variable for us:

```
(11_env)learning_log$ heroku config:set DEBUG='FALSE'  
Setting DEBUG and restarting ⬤ learning-log... done, v7  
DEBUG: FALSE  
(11_env)learning_log$
```

Whenever you set an environment variable on Heroku, it automatically restarts the project so the environment variable can take effect.

To check that the deployment is more secure now, enter your project's URL with a path we haven't defined. For example, try to visit <http://learning-log.herokuapp.com/letmein/>. You should see a generic error page on your live deployment that doesn't give away any specific information about the project. If you try the same request on the local version of Learning Log at <http://localhost:8000/letmein/>, you should see the full Django error page. The result is perfect: you'll see informative error messages when you're developing the project further on your own system. But users on the live site won't see critical information about the project's code.

If you're just deploying an app and you're troubleshooting the initial deployment, you can run `heroku config:set DEBUG='TRUE'` and temporarily see a full error report on the live site. Just make sure you reset the value to 'FALSE' once you've finished troubleshooting. Also, be careful not to do this once users are regularly accessing your site.

Creating Custom Error Pages

In Chapter 19, we configured Learning Log to return a 404 error if the user requests a topic or entry that doesn't belong to them. You've probably seen some 500 server errors (internal errors) by this point as well. A 404 error usually means your Django code is correct, but the object being requested doesn't exist; a 500 error usually means there's an error in the code you've written, such as an error in a function in *views.py*. Currently, Django returns the same generic error page in both situations. But we can write our own 404 and 500 error page templates that match Learning Log's overall appearance. These templates must go in the root template directory.

Making Custom Templates

In the outermost *learning_log* folder, make a new folder called *templates*. Then make a new file called *404.html*; the path to this file should be *learning_log/templates/404.html*. Here's the code for this file:

```
404.html  
{% extends "learning_logs/base.html" %}  
  
{% block page_header %}  
    <h2>The item you requested is not available. (404)</h2>  
{% endblock page_header %}
```

This simple template provides the generic 404 error page information but is styled to match the rest of the site.

Make another file called *500.html* using the following code:

```
500.html
{%
    extends "learning_logs/base.html"
}

{% block page_header %}
    <h2>There has been an internal error. (500)</h2>
{% endblock page_header %}
```

These new files require a slight change to *settings.py*.

```
settings.py
--snip--
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        --snip--
    },
]
--snip--
```

This change tells Django to look in the root template directory for the error page templates.

Viewing the Error Pages Locally

If you want to see what the error pages look like on your system before pushing them to Heroku, you'll first need to set `Debug=False` on your local settings to suppress the default Django debug pages. To do so, make the following change to *settings.py* (make sure you're working in the part of *settings.py* that applies to the local environment, not the part that applies to Heroku):

```
settings.py
--snip--
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = False
--snip--
```

Now request a topic or entry that doesn't belong to you to see the 404 error page. To test the 500 error page, request a topic or entry that doesn't exist. For example, the URL `http://localhost:8000/topics/999/` should generate a 500 error unless you've generated 999 example topics already!

When you're finished checking the error pages, set the local value of `DEBUG` back to `True` to further develop Learning Log. (Make sure you don't change the way `DEBUG` is handled in the section that manages settings in the Heroku environment.)

NOTE

The 500 error page won't show any information about the user who's logged in, because Django doesn't send any context information in the response when there's a server error.

Pushing the Changes to Heroku

Now we need to commit the error page changes we just made, and push them live to Heroku:

```
❶ (ll_env)learning_log$ git add .
❷ (ll_env)learning_log$ git commit -am "Added custom 404 and 500 error pages."
  3 files changed, 15 insertions(+), 10 deletions(-)
   create mode 100644 templates/404.html
   create mode 100644 templates/500.html
❸ (ll_env)learning_log$ git push heroku master
--snip--
remote: Verifying deploy.... done.
To https://git.heroku.com/learning-log.git
  d5075a1..4bd3b1c master -> master
(ll_env)learning_log$
```

We issue the `git add .` command at ❶ because we created some new files in the project, so we need to tell Git to start tracking these files. Then we commit the changes ❷ and push the updated project to Heroku ❸.

Now when an error page appears, it should have the same styling as the rest of the site, making for a smoother user experience when errors arise.

Using the `get_object_or_404()` Method

At this point, if a user manually requests a topic or entry that doesn't exist, they'll get a 500 server error. Django tries to render the nonexistent page, but it doesn't have enough information to do so, and the result is a 500 error. This situation is more accurately handled as a 404 error, and we can implement this behavior using the Django shortcut function `get_object_or_404()`. This function tries to get the requested object from the database, but if that object doesn't exist, it raises a 404 exception. We'll import this function into `views.py` and use it in place of `get()`:

```
views.py
from django.shortcuts import render, redirect, get_object_or_404
from django.contrib.auth.decorators import login_required
--snip--
@login_required
def topic(request, topic_id):
    """Show a single topic and all its entries."""
    topic = get_object_or_404(Topic, id=topic_id)
    # Make sure the topic belongs to the current user.
    --snip--
```

Now when you request a topic that doesn't exist (for example, `http://localhost:8000/topics/999/`), you'll see a 404 error page. To deploy this change, make a new commit and then push the project to Heroku.

Ongoing Development

You might want to further develop Learning Log after your initial push to a live server or develop your own projects to deploy. There's a fairly consistent process for updating projects.

First, you'll make any changes needed to your local project. If your changes result in any new files, add those files to the Git repository using the command `git add .` (be sure to include the dot at the end of the command). Any change that requires a database migration will need this command, because each migration generates a new migration file.

Second, commit the changes to your repository using `git commit -am "commit message"`. Then push your changes to Heroku using the command `git push heroku master`. If you migrated your database locally, you'll need to migrate the live database as well. You can either use the one-off command `heroku run python manage.py migrate`, or open a remote terminal session with `heroku run bash` and run the command `python manage.py migrate`. Then visit your live project, and make sure the changes you expect to see have taken effect.

It's easy to make mistakes during this process, so don't be surprised when something goes wrong. If the code doesn't work, review what you've done and try to spot the mistake. If you can't find the mistake or you can't figure out how to undo the mistake, refer to the suggestions for getting help in Appendix C. Don't be shy about asking for help: everyone else learned to build projects by asking the same questions you're likely to ask, so someone will be happy to help you. Solving each problem that arises helps you steadily develop your skills until you're building meaningful, reliable projects and you're answering other people's questions as well.

The SECRET_KEY Setting

Django uses the value of the `SECRET_KEY` setting in `settings.py` to implement a number of security protocols. In this project, we've committed our settings file to the repository with the `SECRET_KEY` setting included. This is fine for a practice project, but the `SECRET_KEY` setting should be handled more carefully for a production site. If you build a project that's getting meaningful use, make sure you research how to handle your `SECRET_KEY` setting more securely.

Deleting a Project on Heroku

It's great practice to run through the deployment process a number of times with the same project or with a series of small projects to get the hang of deployment. But you'll need to know how to delete a project that's been deployed. Heroku also limits the number of projects you can host for free, and you don't want to clutter your account with practice projects.

Log in to the Heroku website (<https://heroku.com/>); you'll be redirected to a page showing a list of your projects. Click the project you want to delete. You'll see a new page with information about the project. Click the **Settings**

link, and scroll down until you see a link to delete the project. This action can't be reversed, so Heroku will ask you to confirm the request for deletion by manually entering the project's name.

If you prefer working from a terminal, you can also delete a project by issuing the `destroy` command:

```
(ll_env)learning_log$ heroku apps:destroy --app appname
```

Here, `appname` is the name of your project, which is either something like `secret-lowlands-82594` or `learning-log` if you've renamed the project. You'll be prompted to reenter the project name to confirm the deletion.

NOTE

Deleting a project on Heroku does nothing to your local version of the project. If no one has used your deployed project and you're just practicing the deployment process, it's perfectly reasonable to delete your project on Heroku and redeploy it.

TRY IT YOURSELF

20-3. Live Blog: Deploy the Blog project you've been working on to Heroku. Make sure you set `DEBUG` to `False`, so users don't see the full Django error pages when something goes wrong.

20-4. More 404s: The `get_object_or_404()` function should also be used in the `new_entry()` and `edit_entry()` views. Make this change, test it by entering a URL like `http://localhost:8000/new_entry/999/`, and check that you see a 404 error.

20-5. Extended Learning Log: Add one feature to Learning Log, and push the change to your live deployment. Try a simple change, such as writing more about the project on the home page. Then try adding a more advanced feature, such as giving users the option of making a topic public. This would require an attribute called `public` as part of the `Topic` model (this should be set to `False` by default) and a form element on the `new_topic` page that allows the user to change a topic from private to public. You'd then need to migrate the project and revise `views.py` so any topic that's `public` is visible to unauthenticated users as well. Remember to migrate the live database after you've pushed your changes to Heroku.

Summary

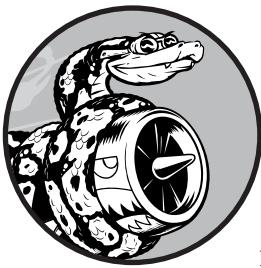
In this chapter, you learned to give your projects a simple but professional appearance using the Bootstrap library and the `django-bootstrap4` app. Using Bootstrap, the styles you choose will work consistently on almost any device people use to access your project.

You learned about Bootstrap's templates and used the `Navbar static` template to create a simple look and feel for Learning Log. You used a jumbotron to make a home page's message stand out and learned to style all the pages in a site consistently.

In the final part of the project, you learned how to deploy a project to Heroku's servers so anyone can access it. You made a Heroku account and installed some tools that help manage the deployment process. You used Git to commit the working project to a repository and then pushed the repository to Heroku's servers. Finally, you learned to begin securing your app by setting `DEBUG=False` on the live server.

Now that you've finished Learning Log, you can start building your own projects. Start simple, and make sure the project works before adding complexity. Enjoy your continued learning, and good luck with your projects!

AFTERWORD



Congratulations! You've learned the basics of Python and applied your knowledge to meaningful projects. You've made a game, visualized some data, and made a web application. From here, you can go in a number of different directions to continue developing your programming skills.

First, you should continue to work on meaningful projects that interest you. Programming is more appealing when you're solving relevant and significant problems, and you now have the skills to engage in a variety of projects. You could invent your own game or write your own version of a classic arcade game. You might want to explore some data that's important to you and make visualizations that show interesting patterns and connections. You could create your own web application or try to emulate one of your favorite apps.

Whenever possible, invite other people to try using your programs. If you write a game, let other people play it. If you make a visualization, show it to others and see if it makes sense to them. If you make a web app, deploy it online and invite others to try it out. Listen to your users and try to incorporate their feedback into your projects; you'll become a better programmer if you do.

When you work on your own projects, you'll run into problems that are challenging, or even impossible, to solve on your own. Keep finding ways to ask for help, and find your own place in the Python community. Join a local Python User Group or explore some online Python communities. Consider attending a PyCon near you as well.

You should strive to maintain a balance between working on projects that interest you and developing your Python skills in general. Many Python learning sources are available online, and a large number of Python books target intermediate programmers. Many of these resources will be accessible to you now that you know the basics and how to apply your skills. Working through Python tutorials and books will build directly on what you learned here and deepen your understanding of programming in general and Python in particular. Then when you go back to working on projects after focusing on learning about Python, you'll be capable of solving a wider variety of problems more efficiently.

Congratulations on how far you've come, and good luck with your continued learning!

A

INSTALLATION AND TROUBLESHOOTING



Python has several versions, and there are a number of ways to set it up on each operating system. Use this appendix to install Python if the approach in Chapter 1 didn't work or if you want to install a different version of Python than the one that came with your system.

Python on Windows

The instructions in Chapter 1 show you how to install Python using the official installer at <https://python.org/>. If you couldn't get Python to run after using the installer, the troubleshooting instructions in this section should help you get Python up and running.

Finding the Python Interpreter

If you've entered the simple command `python` and get an error, such as *python is not recognized as an internal or external command*, you most likely

forgot to select the *Add Python to PATH* option when you ran the installer. In this case, you'll need to tell Windows where to find the Python interpreter. To find it, open your C drive and find the folder that starts with the name *Python* (you might need to enter the word *python* in the Windows Explorer search bar to find the right folder, because it might be nested further down). Open the folder, and look for a file named *python* in lowercase. Right-click this file and choose **Properties**; the path to this file will be listed under the heading *Location*.

To tell Windows where to find the interpreter, open a terminal window and enter the path followed by the `--version` command, like so:

```
$ C:\Python37\python --version
Python 3.7.2
```

Your path might look something more like `C:\Users\username\Programs\Python37\python` on your system. Using this path, Windows should then run the Python interpreter.

Adding Python to Your Path Variable

It's annoying to type the full path each time you want to start a Python terminal session, so let's add the path to the system so you can just use the `python` command. Open your system's **Control Panel**, click **System and Security**, and then click **System**. Click **Advanced System Settings**. In the window that appears, click **Environment Variables**.

In the box labeled *System variables*, look for a variable called *Path*. Click the word **Path**, and then click **Edit**. You should see a list of locations that your system searches through when it looks for programs. Click **New**, and paste the path to your *python.exe* file in the text box that appears. If your system is set up like mine, that would be:

```
C:\Python37
```

Notice that we're not including the name of the *python.exe* file; we're just telling the system where to look for it.

Close your terminal window and open a new one. Doing so will load the new *Path* variable into your terminal session. Now when you enter `python --version`, you should see the version of Python you just added to your *Path* variable. You can now start a Python terminal session by just entering `python` at a command prompt.

NOTE

*If you're using an earlier version of Windows, you might see a box labeled *Variable value* when you click **Edit**. If you see this box, use the right arrow key to scroll all the way to the right. Be careful not to overwrite the existing variable; if you do, click **Cancel** and try again. Add a semicolon and the path to your *python.exe* file to the existing variable:*

```
%SystemRoot%\system32\...\System32\WindowsPowerShell\v1.0\;C:\Python37
```

Reinstalling Python

If you’re still unable to run Python, oftentimes uninstalling Python and running the installer again will address any problems that occurred in your first attempt.

To do this, open your system’s **Control Panel** and click **Programs and Features**. Scroll down until you see the version of Python you just installed, and select it. Click **Uninstall/Change**, and then click **Uninstall** in the dialog that appears. Then run the installer again using the instructions in Chapter 1, but this time make sure you select the **Add Python to PATH** option and any other settings that are relevant to your system. If you’re still running into trouble and aren’t sure where to get help, see the suggestions in Appendix C.

Python on macOS

The installation instructions in Chapter 1 use the official Python installer at <https://python.org/>, which I recommend you use unless you have a specific reason not to. Another approach uses Homebrew, a tool you can use to install a variety of software on macOS. If you’re already using Homebrew and want to use it to install Python, or if the people you’re working with use Homebrew and you want a similar setup to what they’re using, you can use the following instructions.

Installing Homebrew

Homebrew depends on some of the command line tools from Apple’s Xcode package, so you’ll first need to install the Xcode command line tools. Open a terminal and run this command:

```
$ xcode-select --install
```

Click through the confirmation dialogs that appear (this might take a while, depending on your connection speed). Next, install Homebrew by entering the following command:

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

You can find this command at <https://brew.sh/>. Make sure you include a space between curl -fsSL and the URL.

NOTE

The -e in this command tells Ruby (the programming language Homebrew is written in) to execute the code that’s downloaded here. You should only run commands like this from sources you trust.

To confirm that Homebrew installed correctly, run this command:

```
$ brew doctor
Your system is ready to brew.
```

This output means you’re ready to use Homebrew to install packages to your system.

Installing Python

To install the latest version of Python, enter the following command:

```
$ brew install python
```

Check which version was installed by using this command:

```
$ python3 --version
Python 3.7.2
$
```

Now you can start a Python terminal session using the command `python3`. You can also use the `python3` command in your text editor so it runs programs with the version of Python you just installed instead of the system’s earlier version. If you need help configuring Sublime Text to use the version you just installed, see the instructions in Chapter 1.

Python on Linux

Python is included by default on almost every Linux system. But if the default version is earlier than Python 3.6, you should install the latest version. The following instructions should work for most apt-based systems.

We’ll use a package called `deadsnakes`, which makes it easy to install multiple versions of Python. Enter the following commands:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt-get update
$ sudo apt install python3.7
```

These commands should install Python 3.7 onto your system.

Enter the following command to start a terminal session that runs Python 3.7:

```
$ python3.7
>>>
```

You’ll also want to use this command when you configure your text editor and when you run programs from the terminal.

Python Keywords and Built-in Functions

Python comes with its own set of keywords and built-in functions. It's important to be aware of these when you're naming variables: your variable names cannot be the same as these keywords and shouldn't be the same as the function names, or you'll overwrite the functions.

In this section, we'll list Python's keywords and built-in function names, so you'll know which names to avoid.

Python Keywords

Each of the following keywords has a specific meaning, and you'll see an error if you try to use any of them as a variable name.

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

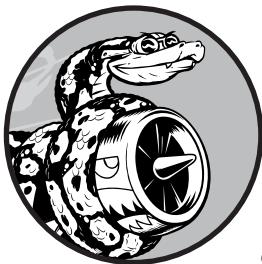
Python Built-in Functions

You won't get an error if you use one of the following readily available built-in functions as a variable name, but you'll override the behavior of that function:

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

B

TEXT EDITORS AND IDEs



Programmers spend a lot of time writing, reading, and editing code, and using a text editor or an *integrated development environment* (*IDE*) to make this work as efficient as possible is essential. A good editor will do simple tasks, like highlight your code's structure so you can catch common bugs as you're working. But it won't do so much that it distracts you from your thinking. Editors also have useful features like automatic indenting, markers to show appropriate line length, and keyboard shortcuts for common operations.

An IDE is a text editor with a number of other tools included, like interactive debuggers and code introspection. An IDE examines your code as you enter it and tries to learn about the project you're building. For example, when you start typing the name of a function, an IDE might show you all the arguments that function accepts. This behavior can be very helpful when everything works and you understand what you're seeing. But it can also be overwhelming as a beginner and difficult to troubleshoot when you aren't sure why your code isn't working in the IDE.

I encourage you to use a simple text editor while you’re learning to code. Text editors also put a much lighter load on your system; so if you’re working on an older machine or one with fewer resources, a text editor will work better than an IDE. If you’re already familiar with IDEs or if people around you use an IDE and you want to use a similar environment, by all means try them out.

Don’t worry too much about choosing your tools at this point; your time will be better spent digging into the language and working on the projects you’re interested in. Once you’ve mastered the basics, you’ll have a better idea what tools work for you.

In this appendix, we’ll set up the Sublime Text text editor to help you work more efficiently. We’ll also take a brief look at a number of other editors you might consider using or see other Python programmers using.

Customizing Sublime Text Settings

In Chapter 1, you configured Sublime Text to use the Python version you want it to when running your programs. Now we’ll configure it to do some of the tasks mentioned at the beginning of this appendix.

Converting Tabs to Spaces

If you use a mix of tabs and spaces in your code, it can cause problems in your programs that are difficult to diagnose. To avoid this, you can configure Sublime Text to always use spaces for indentation, even when you press the TAB key. Go to **View ▶ Indentation** and make sure the **Indent Using Spaces** option is selected. If it’s not, select it. Also, make sure the **Tab Width** is set to 4 spaces.

If you already have a mix of tabs and spaces in one of your programs, you can convert all tabs to spaces by clicking **View ▶ Indentation ▶ Convert Tabs to Spaces**. You can also access these settings by clicking **Spaces** at the bottom right of the Sublime Text window.

Now you can use the TAB key to indent lines of code, and Sublime Text will insert spaces automatically to indent those lines.

Setting the Line Length Indicator

Most editors allow you to set up a visual cue, usually a vertical line, to show where your lines should end. In the Python community, the convention is to limit lines to 79 characters or less. To set this feature, select **View ▶ Ruler**, and then click **80**. Sublime Text will place a vertical line at the 80-character mark to help restrict your code lines to an appropriate length.

Indenting and Unindenting Code Blocks

To indent an entire block of code, highlight it and select **Edit ▶ Line ▶ Indent** or press **CTRL-[**, or **⌘-[** on macOS. To unindent a block of code, click **Edit ▶ Line ▶ Unindent** or press **CTRL-[,** or **⌘-[,**.

Commenting Out Blocks of Code

To temporarily disable a block of code, you can highlight the block and comment it so Python will ignore it. Select **Edit ▶ Comment ▶ Toggle Comment** (CTRL-/ or ⌘-/). The selected lines will be commented out with a hash mark (#) indented at the same level as the line of code to indicate these are not regular comments. When you want to uncomment the block of code, highlight the block and reissue the same command.

Saving Your Configuration

Some of the settings mentioned only affect the current file you’re working in. To make your settings affect all files you open in Sublime Text, you’ll need to define your user settings. Select **Sublime Text ▶ Preferences ▶ Settings**, and look for the file *Preferences.sublime-settings – User*. Enter the following in this file:

```
{  
    "rulers": [80],  
    "translate_tabs_to_spaces": true  
}
```

Save this file, and your ruler and tab settings will apply to all files you work with in Sublime Text. If you add more settings to this file, make sure each line ends with a comma except the last line. You can look at other users’ settings files online and customize your editor to the settings that help you work most efficiently.

Further Customizations

You can customize Sublime Text in many ways to help you work even more efficiently. As you’re exploring the menus, keep an eye out for keyboard shortcuts for the menu items you use most often. Every time you use a keyboard shortcut instead of reaching for the mouse or trackpad, you become a bit more efficient. Don’t try to learn everything at once; just try to become efficient with the actions you use most, and be on the lookout for other features that might help you develop your own workflow.

Other Text Editors and IDEs

You’ll hear about and see people using a number of other text editors. Most of them can be configured to help you in the same way you customized Sublime Text. Here’s a small selection of text editors you might hear about.

IDLE

IDLE is a text editor that’s included with Python. It’s a little less intuitive to work with than Sublime Text, but you’ll see references to it in other tutorials aimed at beginners, so you might want to give it a try.

Geany

Geany is a simple text editor that lets you run all of your programs directly from the editor. It displays all of your output in a terminal window, which helps you become comfortable using terminals. Geany has a very simple interface, but it's powerful enough that a significant number of experienced programmers still use it.

Emacs and Vim

Emacs and Vim are two popular editors favored by many experienced programmers because they're designed to be used so your hands never have to leave the keyboard. This makes writing, reading, and modifying code very efficient once you learn how the editor works. It also means both editors have a fairly steep learning curve. Vim is included on most Linux and macOS machines, and both Emacs and Vim can be run entirely inside a terminal. For this reason, they're often used to write code on servers through a remote terminal session.

Programmers will often recommend that you give them a try. But many proficient programmers forget how much new programmers are already trying to learn. It's beneficial to be aware of these editors, but hold off on using them until you're comfortable working with code in a simpler editor that lets you focus on learning to program rather than learning to use an editor.

Atom

Atom is a text editor with some features that you'd normally find in an IDE. You can open an individual file you're working on, or you can open a project folder and Atom will instantly make all the files in that project easily accessible. Atom is integrated with Git and GitHub, so as you start to use version control you'll be able to work with local and remote repositories from within your editor instead of having to do so in a separate terminal.

Atom allows you to install packages as well, so you can extend its behavior in many ways. A number of these packages incorporate behavior that makes Atom behave more like an IDE.

Visual Studio Code

Visual Studio Code, also called VS Code, is another editor that acts more like an IDE. VS Code supports efficient use of a debugger, has integrated version control support, and also offers code completion tools.

PyCharm

PyCharm is a popular IDE among Python programmers because it was built to work specifically with Python. The full version requires a paid subscription, but a free version called the PyCharm Community Edition is also available that many developers find useful.

PyCharm features a *linter*, which checks that your coding style matches common Python conventions, and offers suggestions when you deviate from normal Python formatting. It also has an integrated debugger to help you resolve errors proficiently and modes that help you work efficiently with a number of popular Python libraries.

Jupyter Notebooks

Jupyter Notebook is a different kind of tool than traditional text editors or IDEs in that it's a web app primarily built of blocks; each block is either a code block or a text block. The text blocks are rendered in Markdown, so you can include simple formatting in your text blocks.

Jupyter Notebooks were developed to support the use of Python in scientific applications, but they have since expanded to become useful in a wide variety of situations. Rather than just writing comments inside a `.py` file, you can write clear text with simple formatting, such as headers, bulleted lists, and hyperlinks in between sections of code. Every code block can be run independently, allowing you to test small pieces of your program, or you can run all the code blocks at once. Each code block has its own output area, and you can toggle the output areas on or off as needed.

Jupyter Notebooks can be confusing at times because of the interactions between different cells. If you define a function in one cell, that function is available to other cells as well. This is beneficial most of the time, but it can be confusing in longer notebooks and if you don't fully understand how the Notebook environment works.

If you're doing any scientific or data-focused work in Python, you'll almost certainly see Jupyter Notebooks at some point.

C

GETTING HELP



Everyone gets stuck at some point when they're learning to program. So one of the most important skills to learn as a programmer is how to get unstuck efficiently. This appendix outlines several ways to help you get going again when programming gets confusing.

First Steps

When you're stuck, your first step should be to assess your situation. Before you ask for help from anyone else, answer the following three questions clearly:

- What are you trying to do?
- What have you tried so far?
- What results have you been getting?

Make your answers as specific as possible. For the first question, explicit statements like “I’m trying to install the latest version of Python on my Windows 10 laptop” are detailed enough for others in the Python community to help you. Statements like “I’m trying to install Python” don’t provide enough information for others to offer much help.

Your answer to the second question should provide enough detail so you won’t be advised to repeat what you’ve already tried: “I went to <https://python.org/downloads/> and clicked the Download button for my system. Then I ran the installer,” is more helpful than “I went to the Python website and downloaded something.”

For the third question, it’s helpful to know the exact error messages you received so you can search online for a solution or provide them when asking for help.

Sometimes just answering these three questions before you ask for help from others allows you to see something you’re missing and get you unstuck without having to go any further. Programmers even have a name for this: it’s called *rubber duck debugging*. The idea is that if you clearly explain your situation to a rubber duck (or any inanimate object), and ask it a specific question, you’ll often be able to answer your own question. Some programming shops even keep a real rubber duck around to encourage people to “talk to the duck.”

Try It Again

Just going back to the start and trying again can be enough to solve many problems. Say you’re trying to write a for loop based on an example in this book. You might have only missed something simple, like a colon at the end of the for line. Going through the steps again might help you avoid repeating the same mistake.

Take a Break

If you’ve been working on the same problem for a while, taking a break is one of the best tactics you can try. When we work on the same task for long periods of time, our brains start to zero in on only one solution. We lose sight of the assumptions we’ve made, and taking a break helps us get a fresh perspective on the problem. It doesn’t need to be a long break, just something that gets you out of your current mind-set. If you’ve been sitting for a long time, do something physical: take a short walk or go outside for a bit; perhaps drink a glass of water or eat a light and healthy snack.

If you’re getting frustrated, it might be worth putting your work away for the day. A good night’s sleep almost always makes a problem more approachable.

Refer to This Book’s Resources

The online resources for this book, available at <https://nostarch.com/pythoncrashcourse2e/>, include a number of helpful sections about setting

up your system and working through each chapter. If you haven’t done so already, take a look at these resources and see if there’s anything that helps your situation.

Searching Online

Chances are that someone else has had the same problem you’re having and has written about it online. Good searching skills and specific inquiries will help you find existing resources to solve the issue you’re facing. For example, if you’re struggling to install the latest version of Python on Windows 10, searching for *install python windows 10* and limiting the results to resources from the last year might direct you to a clear answer.

Searching the exact error message can be extremely helpful too. For example, say you get the following error when you try to start a Python terminal session:

```
> python
'python' is not recognized as an internal or external command,
operable program or batch file
>
```

Searching for the full phrase “python is not recognized as an internal or external command” will probably yield some good advice.

When you start searching for programming-related topics, a few sites will appear repeatedly. I’ll describe some of these sites briefly, so you’ll know how helpful they’re likely to be.

Stack Overflow

Stack Overflow (<https://stackoverflow.com/>) is one of the most popular question-and-answer sites for programmers, and will often appear in the first page of results on Python-related searches. Members post questions when they’re stuck, and other members try to give helpful responses. Users can vote for the responses they find most helpful, so the best answers are usually the first ones you’ll find.

Many basic Python questions have very clear answers on Stack Overflow, because the community has refined them over time. Users are encouraged to post updates too, so responses tend to stay relatively current. At the time of this writing, over one million Python-related questions have been answered on Stack Overflow.

The Official Python Documentation

The official Python documentation (<https://docs.python.org/>) is a bit more hit or miss for beginners, because its purpose is more to document the language than to provide explanations. The examples in the official documentation should work, but you might not understand everything shown. Still, it’s a good resource to check when it comes up in your searches and will become more useful to you as you continue building your understanding of Python.

Official Library Documentation

If you’re using a specific library, such as Pygame, Matplotlib, Django, and so on, links to the official documentation for that project will often appear in searches—for example, <https://docs.djangoproject.com/> is very helpful. If you’re planning to work with any of these libraries, it’s a good idea to become familiar with their official documentation.

r/learnpython

Reddit is made up of a number of subforums called *subreddits*. The *r/learnpython* subreddit (<https://reddit.com/r/learnpython/>) is fairly active and supportive. Here you can read others’ questions and post your own.

Blog Posts

Many programmers maintain blogs and share posts about the parts of the language they’re working with. You should skim the first few comments on a blog post to see what reactions other people have had before taking any advice. If no comments appear, take the post with a grain of salt. It’s possible no one else has verified the advice.

Internet Relay Chat

Many programmers interact in real time through Internet Relay Chat (IRC). If you’re stuck on a problem and searching online isn’t providing answers, asking in an IRC channel might be a good option. Most people who hang out in these channels are polite and helpful, especially if you can be specific about what you’re trying to do, what you’ve already tried, and what results you’re getting.

Making an IRC Account

To create an account on IRC, go to <https://webchat.freenode.net/>. Choose a nickname, fill out the CAPTCHA box, and click **Connect**. You’ll see a message welcoming you to the freenode IRC server. In the box at the bottom of the window, enter the following command:

```
/msg nickserv register password email
```

Enter your own password and email address in place of *password* and *email*. Choose a password that you don’t use for any other account. You’ll receive an email with instructions to verify your account. The email will provide you with a command like this:

```
/msg nickserv verify register nickname verification_code
```

Paste this line into the IRC site with *nickname* as the name you chose earlier and a value for *verification_code*. Now you’re ready to join a channel.

If you have trouble logging into your account at some point, you can issue the following command:

```
/msg nickserv identify nickname password
```

Replace *nickname* and *password* with your own nickname and password. This will authenticate you on the network, and you'll be able to access channels that require an authenticated nickname.

Channels to Join

To join the main Python channel, enter `/join #python` in the input box. You'll see a confirmation that you joined the channel and some general information about the channel.

The channel `##learnpython` (with two hashtags) is usually quite active as well. This channel is associated with <https://reddit.com/r/learnpython/>, so you'll see messages about posts on *r/learnpython* too. You might want to join the `#django` channel if you're working on web applications.

After you've joined a channel, you can read the conversations other people are having and ask your own questions as well.

IRC Culture

To get effective help, you should know a few details about IRC culture. Focusing on the three questions at the beginning of this appendix will definitely help guide you to a successful solution. People will be happy to help you if you can explain precisely what you're trying to do, what you've already tried, and the exact results you're getting. If you need to share code or output, IRC members use external sites made for this purpose, such as <https://bpaste.net/+python>. (This is where `#python` sends you to share code and output.) This keeps the channels from being flooded with code and also makes it much easier to read the code that people share.

Being patient will always make people more likely to help you. Ask your question concisely, and then wait for someone to respond. Often, people are in the middle of many conversations, but usually someone will address you in a reasonable amount of time. If few people are in the channel, it might take a while to get a response.

Slack

Slack is like a modern-day reinvention of IRC. It's often used for internal company communications, but there are also many public groups you can join. If you want to check out Python Slack groups, start with <https://pyslackers.com/>. Click the **Slack** link at the top of the page, and enter your email address to get an invitation.

Once you're in the Python Developers workspace, you'll see a list of channels. Click **Channels**, and then choose the topics that interest you. You might want to start with the `#learning_python` and `#django` channels.

Discord

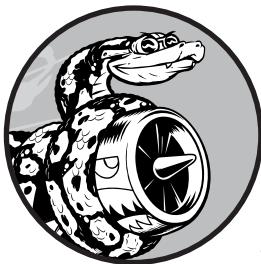
Discord is another online chat environment with a Python community where you can ask for help and follow Python-related discussions.

To check it out, head to <https://pythondiscord.com/> and click the **Chat Now** link. You should see a screen with an automatically generated invitation; click **Accept Invite**. If you already have a Discord account, you can log in with your existing account. If you don't have an account, enter a username and follow the prompts to complete your Discord registration.

If this is your first time visiting the Python Discord, you'll need to accept the rules for the community before participating fully. Once you've done that, you can join any of the channels that interest you. If you're looking for help, be sure to post in one of the Python Help channels.

D

USING GIT FOR VERSION CONTROL



Version control software allows you to take snapshots of a project whenever it's in a working state. When you make changes to a project—for example, when you implement a new feature—you can revert back to a previous working state if the project's current state isn't functioning well.

Using version control software gives you the freedom to work on improvements and make mistakes without worrying about ruining your project. This is especially critical in large projects, but can also be helpful in smaller projects, even when you're working on programs contained in a single file.

In this appendix, you'll learn to install Git and use it for version control in the programs you're working on now. Git is the most popular version control software in use today. Many of its advanced tools help teams collaborate on large projects, but its most basic features also work well for solo developers. Git implements version control by tracking the changes made to every file in a project; if you make a mistake, you can just return to a previously saved state.

Installing Git

Git runs on all operating systems, but there are different approaches to installing it on each system. The following sections provide specific instructions for each operating system.

Installing Git on Windows

You can download an installer for Git at <https://git-scm.com/>. You should see a download link for an installer that's appropriate for your system.

Installing Git on macOS

Git might already be installed on your system, so try issuing the command `git --version`. If you see output listing a specific version number, Git is installed on your system. If you see a message prompting you to install or update Git, simply follow the onscreen directions.

You can also visit <https://git-scm.com/>, where you should see a download link for an appropriate installer for your system.

Installing Git on Linux

To install Git on Linux, enter the following command:

```
$ sudo apt install git-all
```

That's it. You can now use Git in your projects.

Configuring Git

Git keeps track of who makes changes to a project, even when only one person is working on the project. To do this, Git needs to know your username and email. You must provide a username, but you can make up a fake email address:

```
$ git config --global user.name "username"  
$ git config --global user.email "username@example.com"
```

If you forget this step, Git will prompt you for this information when you make your first commit.

Making a Project

Let's make a project to work with. Create a folder somewhere on your system called `git_practice`. Inside the folder, make a simple Python program:

```
hello_git.py      print("Hello Git world!")
```

We'll use this program to explore Git's basic functionality.

Ignoring Files

Files with the extension `.pyc` are automatically generated from `.py` files, so we don't need Git to keep track of them. These files are stored in a directory called `__pycache__`. To tell Git to ignore this directory, make a special file called `.gitignore`—with a dot at the beginning of the filename and no file extension—and add the following line to it:

```
.gitignore  
__pycache__/
```

This file tells Git to ignore any file in the `__pycache__` directory. Using a `.gitignore` file will keep your project clutter free and easier to work with.

You might need to modify your text editor's settings so it will show hidden files in order to open `.gitignore`. Some editors are set to ignore filenames that begin with a dot.

Initializing a Repository

Now that you have a directory containing a Python file and a `.gitignore` file, you can initialize a Git repository. Open a terminal, navigate to the `git_practice` folder, and run the following command:

```
git_practice$ git init  
Initialized empty Git repository in git_practice/.git/  
git_practice$
```

The output shows that Git has initialized an empty repository in `git_practice`. A *repository* is the set of files in a program that Git is actively tracking. All the files Git uses to manage the repository are located in the hidden directory `.git`, which you won't need to work with at all. Just don't delete that directory, or you'll lose your project's history.

Checking the Status

Before doing anything else, let's look at the project's status:

```
git_practice$ git status  
❶ On branch master  
  
No commits yet  
  
❷ Untracked files:  
(use "git add <file>..." to include in what will be committed)  
  
.gitignore  
hello_git.py  
  
❸ nothing added to commit but untracked files present (use "git add" to track)  
git_practice$
```

In Git, a *branch* is a version of the project you’re working on; here you can see that we’re on a branch named `master` ❶. Each time you check your project’s status, it should show that you’re on the branch `master`. You then see that we’re about to make the initial commit. A *commit* is a snapshot of the project at a particular point in time.

Git informs us that untracked files are in the project ❷, because we haven’t told it which files to track yet. Then we’re told that there’s nothing added to the current commit, but untracked files are present that we might want to add to the repository ❸.

Adding Files to the Repository

Let’s add the two files to the repository, and check the status again:

```
❶ git_practice$ git add .
❷ git_practice$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

❸ new file:  .gitignore
new file:  hello_git.py

git_practice$
```

The command `git add .` adds all files within a project that aren’t already being tracked to the repository ❶. It doesn’t commit the files; it just tells Git to start paying attention to them. When we check the status of the project now, we can see that Git recognizes some changes that need to be committed ❷. The label *new file* means these files were newly added to the repository ❸.

Making a Commit

Let’s make the first commit:

```
❶ git_practice$ git commit -m "Started project."
❷ [master (root-commit) ee76419] Started project.
❸ 2 files changed, 4 insertions(+)
   create mode 100644 .gitignore
   create mode 100644 hello_git.py
❹ git_practice$ git status
On branch master
nothing to commit, working tree clean
git_practice$
```

We issue the command `git commit -m "message"` ❶ to take a snapshot of the project. The `-m` flag tells Git to record the message that follows ("Started project.") in the project's log. The output shows that we're on the `master` branch ❷ and that two files have changed ❸.

When we check the status now, we can see that we're on the `master` branch, and we have a clean working tree ❹. This is the message you want to see each time you commit a working state of your project. If you get a different message, read it carefully; it's likely you forgot to add a file before making a commit.

Checking the Log

Git keeps a log of all commits made to the project. Let's check the log:

```
git_practice$ git log
commit a9d74d87f1aa3b8f5b2688cb586eac1a908cfc7f (HEAD -> master)
Author: Eric Matthes <eric@example.com>
Date:   Mon Jan 21 21:24:28 2019 -0900

    Started project.
git_practice$
```

Each time you make a commit, Git generates a unique, 40-character reference ID. It records who made the commit, when it was made, and the message recorded. You won't always need all of this information, so Git provides an option to print a simpler version of the log entries:

```
git_practice$ git log --pretty=oneline
ee76419954379819f3f2cacaf15103ea900ecb2 (HEAD -> master) Started project.
git_practice$
```

The `--pretty=oneline` flag provides the two most important pieces of information: the reference ID of the commit and the message recorded for the commit.

The Second Commit

To see the real power of version control, we need to make a change to the project and commit that change. Here we'll just add another line to `hello_git.py`:

```
hello_git.py
print("Hello Git world!")
print("Hello everyone.")
```

When we check the status of the project, we'll see that Git has noticed the file that changed:

```
git_practice$ git status
❶ On branch master
```

```
Changes not staged for commit:  
  (use "git add <file>..." to update what will be committed)  
  (use "git checkout -- <file>..." to discard changes in working directory)
```

- ❷ modified: hello_git.py
 - ❸ no changes added to commit (use "git add" and/or "git commit -a")
git_practice\$
-

We see the branch we're working on ❶, the name of the file that was modified ❷, and that no changes have been committed ❸. Let's commit the change and check the status again:

- ❶ git_practice\$ git commit -am "Extended greeting."
[master 51f0fe5] Extended greeting.
 1 file changed, 1 insertion(+), 1 deletion(-)
 - ❷ git_practice\$ git status
On branch master
nothing to commit, working tree clean
 - ❸ git_practice\$ git log --pretty=oneline
51f0fe5884e045b91c12c5449fabf4ad0eef8e5d (HEAD -> master) Extended greeting.
ee76419954379819f3f2cacafdf15103ea900ecb2 Started project.
git_practice\$
-

We make a new commit, passing the `-am` flags when we use the command `git commit` ❶. The `-a` flag tells Git to add all modified files in the repository to the current commit. (If you create any new files between commits, simply reissue the `git add .` command to include the new files in the repository.) The `-m` flag tells Git to record a message in the log for this commit.

When we check the project's status, we see that we once again have a clean working directory ❷. Finally, we see the two commits in the log ❸.

Reverting a Change

Now let's look at how to abandon a change and revert back to the previous working state. First, add a new line to `hello_git.py`:

```
hello_git.py  
print("Hello Git world!")  
print("Hello everyone.")  
  
print("Oh no, I broke the project!")
```

Save and run this file.

We check the status and see that Git notices this change:

```
git_practice$ git status  
On branch master  
Changes not staged for commit:  
  (use "git add <file>..." to update what will be committed)
```

```
(use "git checkout -- <file>..." to discard changes in working directory)
```

```
❶ modified: hello_git.py
```

```
no changes added to commit (use "git add" and/or "git commit -a")
git_practice$
```

Git sees that we modified *hello_git.py* ❶, and we can commit the change if we want to. But this time, instead of committing the change, we'll revert back to the last commit when we knew our project was working. We won't do anything to *hello_git.py*: we won't delete the line or use the Undo feature in the text editor. Instead, enter the following commands in your terminal session:

```
git_practice$ git checkout .
git_practice$ git status
On branch master
nothing to commit, working tree clean
git_practice$
```

The command `git checkout` allows you to work with any previous commit. The command `git checkout .` abandons any changes made since the last commit and restores the project to the last committed state.

When you return to your text editor, you'll see that *hello_git.py* has changed back to this:

```
print("Hello Git world!")
print("Hello everyone.")
```

Although going back to a previous state might seem trivial in this simple project, if we were working on a large project with dozens of modified files, all the files that had changed since the last commit would be reverted. This feature is incredibly useful: you can make as many changes as you want when implementing a new feature, and if they don't work, you can discard them without affecting the project. You don't have to remember those changes and manually undo them. Git does all of that for you.

NOTE

You might have to refresh the file in your editor to see the previous version.

Checking Out Previous Commits

You can check out any commit in your log, not just the most recent, by including the first six characters of the reference ID instead of a dot. By checking out and reviewing an earlier commit, you can then return to the latest commit or abandon your recent work and pick up development from the earlier commit:

```
git_practice$ git log --pretty=oneline
51f0fe5884e045b91c12c5449fabf4ad0eef8e5d (HEAD -> master) Extended greeting.
```

```
ee76419954379819f3f2cacaf15103ea900ecb2 Started project.  
git_practice$ git checkout ee7641  
Note: checking out 'ee7641'.
```

- ❶ You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at ee7641... Started project.  
git_practice$
```

When you check out a previous commit, you leave the master branch and enter what Git refers to as a *detached HEAD* state ❶. *HEAD* is the current committed state of the project; you're *detached* because you've left a named branch (`master`, in this case).

To get back to the `master` branch, you check it out:

```
git_practice$ git checkout master  
Previous HEAD position was ee76419 Started project.  
Switched to branch 'master'  
git_practice$
```

This command brings you back to the `master` branch. Unless you want to work with some more advanced features of Git, it's best not to make any changes to your project when you've checked out an old commit. However, if you're the only one working on a project and you want to discard all of the more recent commits and go back to a previous state, you can reset the project to a previous commit. Working from the `master` branch, enter the following:

- ❶ `git_practice$ git status`
On branch master
nothing to commit, working directory clean
- ❷ `git_practice$ git log --pretty=oneline`
51f0fe5884e045b91c12c5449fabf4ad0eef8e5d (HEAD -> master) Extended greeting.
ee76419954379819f3f2cacaf15103ea900ecb2 Started project.
- ❸ `git_practice$ git reset --hard ee76419`
HEAD is now at ee76419 Started project.
- ❹ `git_practice$ git status`
On branch master
nothing to commit, working directory clean
- ❺ `git_practice$ git log --pretty=oneline`
ee76419954379819f3f2cacaf15103ea900ecb2 (HEAD -> master) Started project.
git_practice\$
-

We first check the status to make sure we're on the `master` branch ❶. When we look at the log, we see both commits ❷. We then issue the `git`

`reset --hard` command with the first six characters of the reference ID of the commit we want to revert to permanently ❸. We check the status again and see we're on the `master` branch with nothing to commit ❹. When we look at the log again, we see that we're at the commit we wanted to start over from ❺.

Deleting the Repository

Sometimes you'll mess up your repository's history and won't know how to recover it. If this happens, first consider asking for help using the methods discussed in Appendix C. If you can't fix it and you're working on a solo project, you can continue working with the files but get rid of the project's history by deleting the `.git` directory. This won't affect the current state of any of the files, but it will delete all commits, so you won't be able to check out any other states of the project.

To do this, either open a file browser and delete the `.git` repository or delete it from the command line. Afterwards, you'll need to start over with a fresh repository to start tracking your changes again. Here's what this entire process looks like in a terminal session:

```
❶ git_practice$ git status
On branch master
nothing to commit, working directory clean
❷ git_practice$ rm -rf .git
❸ git_practice$ git status
fatal: Not a git repository (or any of the parent directories): .git
❹ git_practice$ git init
Initialized empty Git repository in git_practice/.git/
❺ git_practice$ git status
On branch master

No commits yet

Untracked files:
(use "git add <file>..." to include in what will be committed)

.gitignore
hello_git.py

nothing added to commit but untracked files present (use "git add" to track)
❻ git_practice$ git add .
❼ git_practice$ git commit -m "Starting over."
[master (root-commit) 6baf231] Starting over.
 2 files changed, 4 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 hello_git.py
❽ git_practice$ git status
On branch master
nothing to commit, working tree clean
git_practice$
```

We first check the status and see that we have a clean working tree ❶. Then we use the command `rm -rf .git` to delete the `.git` directory (`rmdir /s .git` on Windows) ❷. When we check the status after deleting the `.git` folder, we're told that this is not a Git repository ❸. All the information Git uses to track a repository is stored in the `.git` folder, so removing it deletes the entire repository.

We're then free to use `git init` to start a fresh repository ❹. Checking the status shows that we're back at the initial stage, awaiting the first commit ❺. We add the files and make the first commit ❻. Checking the status now shows us that we're on the new `master` branch with nothing to commit ❾.

Using version control takes a bit of practice, but once you start using it you'll never want to work without it again.

INDEX

Symbols

+ (addition), 26
* (asterisk) operator, 147
{ } (braces), 92, 105
/ (division), 26
** (double asterisk) operator, 149
== (equality operator), 72–73
** (exponent), 26
// (floor division), 260
> (greater than), 75
>= (greater than or equal to), 75
(hash mark), for comments, 29
!= (inequality operator), 74
< (less than), 75
<= (less than or equal to), 75
% (modulo operator), 116–117, 122
* (multiplication), 26
\n (newline), 22
! (not), 74
+= operator, 115
[] (square brackets), 34
- (subtraction), 26
\t (tab), 22

A

addition (+), 26
aliases, 152
alice.py, 197–199
Alien Invasion project. *See also*
Pygame
aliens
 checking edges, 266
 collisions, with bullets,
 268–269, 291–292
 collisions, with ship, 272–275
 controlling fleet
 direction, 266
 creating an alien, 256

creating the fleet, 258–264
dropping the fleet, 267
reaching bottom of
 screen, 276
rebuilding the fleet, 270
bullets, 246–252
collisions, with aliens,
 268–269, 291–292
deleting old, 250
firing, 249
limiting number of, 251
making larger, 270
settings, 247
speeding up, 271
classes
 Alien, 256–258
 Bullet, 247–248
 Button, 280–281
 GameStats, 273
 Scoreboard, 288–289
 Settings, 231
 Ship, 233–235
ending the game, 276
files
 alien_invasion.py, 229
 bullet.py, 247
 button.py, 280
 game_stats.py, 273
 scoreboard.py, 288
 settings.py, 231
 ship.bmp, 233
initializing dynamic
 settings, 286
levels
 adding, 285–287
 modifying speed
 settings, 285
 resetting the speed, 287
planning, 228

Alien Invasion project, *continued*

Play button
 adding, 280–285
 deactivating, 284
 drawing, 281
 hiding the mouse
 cursor, 284
 resetting the game, 283
 starting the game, 283
scoring, 288–300
 all hits, 292
 high score, 294
 increasing point values, 292
 level, 296–298
 number of ships, 298–300
 resetting, 291
 rounding and
 formatting, 293–294
 score attribute, 288
 updating, 291
settings, storing, 231
ship
 adjusting speed, 241–243
 continuous movement,
 239–241
 finding an image, 232
 limiting range, 243
amusement_park.py, 80–83
and keyword, 75
API (application programming
 interface), 359
calls, 359
for GitHub, 371
for Hacker News, 372–375
processing responses, 361–365
rate limits, 365
requesting data, 360
visualizing results, 366–371
apostrophe.py, 24
append() method, 37
application programming interface.
 *See API (application
 programming interface)*
arguments, 131–137. *See also*
 functions: arguments
arithmetic, 26
as keyword, 152
assert methods, 212, 216

asterisk (*) operator, 147
attributes, 159. *See also* classes:
 attributes

B

banned_users.py, 77
bicycles.py, 34–36
.bmp (bitmap) image files, 232
body
 of a function, 130
 of an HTML file, 440
Boolean values, 77, 456
Bootstrap, 438–447
braces {}, 92, 105
built-in functions, 471

C

CamelCase, 181
car.py, 162–179
cars.py, 43–45, 72
cities.py, 121
classes
 attributes, 159
 accessing, 160
 default values, 163
 modifying, 164–166
 creating, 158–162
 importing, 174–179
 all classes from a
 module, 177
 multiple classes, 175–177
 single class, 174–175
inheritance, 167–173
 attributes and methods, 169
 child classes, 167
 __init__() method, 167–169
 instances as attributes,
 170–172
 overriding methods, 170
 parent classes, 167
 subclasses, 168
 super() function, 168
 superclasses, 168
instances, 157
methods, 159
 calling, 160
 __init__() method, 159

modeling real-world objects, 173
multiple instances, 161
naming conventions, 159
objects, 157
styling guidelines, 181
comma-separated value files. *See*
 CSV (comma-separated
 value) files
comment.py, 29
comments, 29–30
conditional tests, 72–77. *See also*
 if statements
confirmed_users.py, 124
constants, 28
counting.py, 118, 122
CSV (comma-separated value) files,
 334–346
 error-checking, 343–345
 parsing headers, 334–335
 reading data, 336

D

data analysis, 305
databases. *See* Django: databases
data visualization, 305. *See also*
 Matplotlib; Plotly
datetime module, 337–339
death_valley_highs_lows.py, 343–345
decorators, 429
default values, 134
 class attributes, 163
 function parameters, 134
def keyword, 130
del statement, 39
dice_visual.py, 328–330
dictionaries
 defining, 92
 empty, 94
 formatting larger, 97
 get() method, 98
 KeyError, 98
 key-value pairs, 92–99
 adding, 93
 removing, 96
 looping through, 99–105
 keys, 101
 keys in order, 103
key-value pairs, 99
values, 104
ordering in, 94
sorting a list of, 374
values
 accessing, 93
 modifying, 95
die.py, 324
die_visual.py, 325–327
dimensions.py, 66–67
Discord, 48
div (HTML), 441
division (/), 26
division_calculator.py, 194–197
Django, 379. *See also* Heroku;
 Learning Log project
admin site, 387–392
registering models, 388, 391
associating data with users, 435
Bootstrap, 438–447
 card, 446
 collapsible navigation, 440
 container element, 443
 django-bootstrap4 app, 438
 HTML headers, 439–440
 jumbotron, 443
 navigation bar, 440–442
 styling forms, 444–445
commands
 createsuperuser, 388
 flush, 433
 makemigrations, 387, 391, 432
 migrate, 383
 shell, 392
 startapp, 385, 421
 startproject, 382
creating a project, 381
databases
 cascading delete, 390
 creating, 382
 foreign keys, 390
 many-to-one
 relationships, 390
 migrating, 383, 391
 queries, 404, 433
 querysets, 392–393, 401
 required (non-nullables)
 fields, 432

Django, *continued*

- databases, *continued*
 - resetting, 433
 - SQLite, 383
- decorators, 429
- deployment. *See* Heroku
- development server, 383, 389
- documentation
 - Django, 379
 - models, 386
 - queries, 394
 - templates, 406
- forms, 410–420
 - action argument, 413
 - cross-site request forgery, 413
 - displaying, 413
 - GET and POST requests, 412
 - `ModelForm`, 410, 414
 - pre-filling with data, 419
 - processing, 412, 416
 - validation, 410
 - widgets, 414
- `get_object_or_404()` method, 460
- hashes (for passwords), 388
- HTML
 - div element, 441
 - main element, 442
 - margin, 443
 - padding, 443
 - span element, 442
- HTTP 404 error, 434
- `INSTALLED_APPS`, 386, 421, 438
- installing, 381
- localhost, 383
- logging out, 424
- login page, 422
 - `@login_required`, 429
 - login template, 422
- mapping URLs, 395–396
- migrating the database,
 - 383, 391
- models, 385–393, 431
- privileges, 387
- projects (vs. apps), 384
- `redirect()` function, 411

registration page, 426–428

release cycle, 381

restricting access to data,

- 433–435

restricting access to pages,

- 428–435

`settings.py`

- `INSTALLED_APPS`, 386, 421, 438
- `LOGIN_URL`, 429
- `SECRET_KEY`, 461

shell, 392, 431

starting an app, 385

starting a new project, 382

static files, 448

styling. *See* Django: Bootstrap

superuser, 387

templates, 397

- anchor tags, 399
- block tags, 399
- context dictionary, 401
- filters, 405
- for loop, 402
- indentation in, 399
- inheritance, 398
- linebreaks filter, 405
- template tags, 399

third-party apps, 438

URLs

- capturing values, 404
- namespaces, 399
- URL patterns, 395–396
 - `url` template tag, 399

user ID values, 431

users

- default login view, 422
- displaying message to logged-in user, 424
- logging in a user, 427
- `UserCreationForm`, 427

versions, 381

views, 396

- retrieving objects, 401, 404

docstrings, 130

dog.py, 158–162

dot notation, 151, 160

double asterisk (**) operator, 149

E

earthquakes. *See* mapping earthquakes
electric_car.py, 168–173
electric_car.py module, 178
`enumerate()` function, 335
environment variables, 456
epoch time, 366
eq_explore_data.py, 348–351
equality operator (`==`), 72–73
eq_world_map.py, 351–357
even_numbers.py, 58
even_or_odd.py, 117
exceptions, 183, 194–202
 deciding which errors to report, 201
 else blocks, 196
 failing silently, 200
 `FileNotFoundException`, 197
 handling, 194
 try-except blocks, 194
 using to prevent crashes, 195
 `ZeroDivisionError`, 194
exponents (`**`), 26

F

favorite_languages.py, 97–98
`FileNotFoundException`, 197
file_reader.py, 184–188
files
 closing, 185
 file paths, 185
 absolute, 186
 relative, 186
 opening, 184
 append mode, 192
 read mode, 192
 write mode, 192
 reading from, 184–190
 entire files, 184–185
 line by line, 187
 making a list of lines, 188
 working with contents, 188
 working with large files, 189

writing to

 by appending, 193
 empty files, 191
 multiple lines, 192

first_numbers.py, 57
flags, 120
floats, 26
foods.py, 63–65

for loops, 49. *See also* dictionaries; lists

formatted_name.py, 138–140

full_name.py, 21–22

functions, 129

 alias (as), 152

arguments, 131–137

 arbitrary keyword, 148

 arbitrary number of, 147

 avoiding errors, 136

 keyword, 133

 lists as, 143–146

 mixing positional and arbitrary, 148

 optional, 138

 order of, 133

 positional, 132–133

built-in, 471

calling, 130–137

 equivalent calls, 135–136

 multiple times, 132

defining, 130

dictionaries, returning, 140

lists in

 modifying, 143–145

 protecting, 145

modules, 150–155

 alias (as), 153

 importing all

 functions (*), 153

 importing entire

 modules, 150

 importing specific

 functions, 152

parameters, default

 values for, 134

passing information to. *See*

 functions: arguments

return values, 137–142

styling, 154

G

games. *See* Alien Invasion project;
Pygame
Geany, 476
GET requests, 412
getting help
Discord, 484
IRC (Internet Relay Chat),
482–483
official Python documentation,
481
online resources, 480
r/learnpython, 482
rubber duck debugging, 480
Slack, 483
Stack Overflow, 481
three main questions, 479
Git, 360, 450
branches, 452, 488
commits, 360, 450
 checking out, 491–493
 making, 452, 457, 460, 488,
 490, 493
configuring, 451, 486
detached HEAD, 492
files
 adding, 452, 460, 488, 493
 ignoring, 451, 487
HEAD, 492
installing, 450, 486
log, checking, 489
repositories, 360
 deleting, 493
 initializing, 452, 487, 493
reverting changes, 490
status, checking, 452, 457,
 487–493
GitHub, 360
greater than ($>$), 75
greater than or equal to (\geq), 75
greeter.py, 114, 130–131
greet_users.py, 143
gunicorn package, 448

H

Hacker News, 372
hash mark (#), for comments, 29
head, of an HTML file, 440
HEAD (Git), 492
Hello World, 9
hello_git.py, 486–491
hello_world.py, 10, 15–19
Heroku, 437. *See also* Django; Git;
 Learning Log project
Bash shell, 454
CLI, installing, 448
commands
 config, 458
 destroy, 462
 login, 453
 open, 453
 ps, 453
 rename, 455
 run, 454
 set, 458
databases, setting up, 454
django-heroku package, 448
documentation, 453
environment variables, setting,
 456–458
error pages, custom, 458–460
free plan, limitations of,
 448, 456
making an account, 448
Procfile, 450
projects
 deleting, 461
 pushing to Heroku,
 452–453, 457
 viewing live, 453
Python runtime, specifying, 449
requirements.txt, 448–449
securing, 456
settings.py, modifying for, 450,
 456, 459
superuser, creating, 454–455
URLs, user-friendly, 455
hidden files, 451

hn_article.py, 372
hn_submissions.py, 373
Homebrew, 469

I

IDE (integrated development environment), 473
IDLE, 475
if statements
 and keyword, 75
 Boolean expressions, 77
 checking for
 equality (==), 72
 equality, ignoring case, 73
 empty lists, 87
 inequality (!=), 74
 items in a list, 76
 items not in a list, 77
 conditional tests, 72–77
 elif statement, 80–84
 else statement, 79
 lists and, 85–88
 numerical comparisons, 74–76
 or keyword, 76
 simple, 78
 styling guidelines, 90
 testing multiple conditions,
 83–84
immutable, 65
`import *`, 153
`import this`, 30
indentation errors, 53–56
index errors, 47
inequality operator (!=), 74
infinite loops, 122
inheritance, 167. *See also* classes:
 inheritance
`input()` function, 114–117
 numerical input, 115–116
 prompts, 114
`insert()` method, 38
IRC (Internet Relay Chat), 482–483
`itemgetter()` function, 374
`items()` method, 100

J

JSON files
 examining data, 347
 geoJSON file format, 349
 `json.dump()` function, 348
 `json.load()` function, 204
jumbotron, 443

K

`keys()` method, 101
key-value pairs, 92–99. *See also*
 dictionaries
keyword arguments, 132, 133. *See
 also* functions
keywords, 471

L

language_survey.py, 217
Learning Log project, 379. *See also*
 Django; Heroku
deployment. *See* Git, Heroku
files
 `404.html`, 458
 `500.html`, 459
 `admin.py`, 388
 `base.html`, 399, 403, 423, 425,
 428, 439–443
 `edit_entry.html`, 419
 `forms.py`, 410, 414
 `.gitignore`, 451
 `index.html`, 397, 443
 `learning_logs/urls.py`, 395,
 401, 404, 411, 415, 418
 `learning_log/urls.py`, 395, 422
 `logged_out.html`, 425
 `login.html`, 422–423, 444
 `models.py`, 385, 390, 431
 `new_entry.html`, 416
 `new_topic.html`, 412
 `Procfile`, 450
 `register.html`, 427
 `requirements.txt`, 448
 `runtime.txt`, 449
 `settings.py`, 386, 421, 429, 450,
 456, 459

Learning Log project, *continued*

files, *continued*

- topic.html*, 404–405, 417, 420
- topics.html*, 402, 405, 413, 445
- users/urls.py*, 422, 426
- users/views.py*, 426
- views.py*, 396, 401, 404, 411, 415, 418, 429, 430, 433, 460

pages

- edit entry, 418–420
- home, 394–398
- login, 422–424
- logout, 424
- new entry, 414–417
- new topic, 410–413
- register, 426
- topic, 403
- topics, 400–403

users app, 421–428

virtual environment, 380–381

writing a specification (spec), 380

`len()` function, 45

less than (<), 75

less than or equal to (≤), 75

Linux

- Hello World, running, 10

Python

- checking installed version, 8
- installing, 470
- troubleshooting
 - installation, 11

Sublime Text, installing, 9

terminal

- running programs from, 12
- starting Python session, 9

lists, 33

- as arguments, 143–146
- copying, 63–65
- elements
 - accessing, 34
 - accessing last, 35
 - adding with `append()`, 37
 - adding with `insert()`, 38
 - identifying unique, 104
 - modifying, 36

removing with `del`, 39

removing with `pop()`, 39–41

removing with `remove()`, 41

empty, 38

`enumerate()` function, 335

for loops, 49–56

- nested, 109, 263

if statements and, 85–88

indentation errors, 53–56

indexes, 35

- errors, 46–47
- negative, 35

`len()` function, 45

list comprehensions, 59

naming, 34

numerical lists, 57–60

`range()` function, 58–59

removing all occurrences of a value, 125

slices, 61–63

sorting, 43–46

- `reverse()` method, 45
- `sorted()` function, 44
- `sort()` method, 43

logical errors, 54

M

macOS

- Hello World, running, 10

Python

- checking installed version, 7
- installing, with Homebrew, 469–470
- installing, official
 - version, 7–8
- troubleshooting
 - installation, 11

Sublime Text, installing, 8

terminal

- running programs from, 12
- starting a Python session, 8

magicians.py, 50–53

magic_number.py, 75

making_pizzas.py, 151–153

mapping earthquakes, 347–357

- building a world map, 351

colorscales, 354

- d**
- data
 - downloading, 347, 358
 - examining JSON data, 347
 - extracting locations, 351
 - extracting magnitudes, 350
 - geoJSON file format, 349
 - hover text, 356
 - latitude-longitude ordering, 349
 - magnitudes, representing, 353
 - Scattergeo chart type, 352
 - Matplotlib, 306–323, 336–347
 - axes
 - axis() method, 313
 - ax variable, 307
 - removing, 321
 - fig variable, 307
 - formatting plots
 - alpha argument, 342
 - built-in styles, 310
 - colormaps, 314
 - custom colors, 314
 - labels, 307–308
 - line thickness, 307
 - shading, 342
 - size, 322
 - gallery, 306
 - installing, 306
 - line graphs, 306
 - plot() method, 307, 309
 - pyplot module, 307
 - saving plots, 315
 - scatter plots, 310–313
 - subplots() function, 307
 - methods, 20, 236–237. *See also* classes
 - modules, 150. *See also* classes: modules; functions: modules
 - modulo operator (%), 116–117, 122
 - motorcycles.py*, 37–42
 - mountain_poll.py*, 126
 - mpl_squares.py*, 306–310
 - multiplication (*), 26
 - my_car.py*, 175
 - my_cars.py*, 177–179
 - my_electric_car.py*, 176
- N**
- name errors, 17
 - name_function.py*, 210–215
 - name.py*, 20
 - names.py*, 210
 - nesting, 106–112
 - depth, 110
 - ictionaries in dictionaries, 110–111
 - ictionaries in lists, 106–108
 - lists in dictionaries, 108–110
 - newline (\n), 22
 - None, 99
 - not (!), 74
 - number_reader.py*, 203
 - numbers, 25–28
 - arithmetic, 26
 - comparisons, 74–76
 - exponents, 26
 - floats, 26
 - floor division (/), 260
 - integers, 26
 - mixing integers and floats, 27
 - order of operations, 26
 - round() function, 294
 - underscores in, 28
 - number_writer.py*, 203
- O**
- object-oriented programming, 157.
 - See also* classes
- P**
- parameters, 131
 - parent classes, 167. *See also* classes: inheritance
 - parrot.py*, 114, 118–121
 - pass statement, 200
 - PEP 8, 68–70, 90, 154
 - person.py*, 140–142
 - pets.py*, 125, 132–136

pip, 228
installing Django, 381
installing Matplotlib, 306
installing Plotly, 324
installing Pygame, 228
installing Requests, 361
pi_string.py, 188–190
pizza.py, 147–148
players.py, 61–63
Plotly, 306
 Bar() class, 327
 data, long format, 353
 formatting plots
 bars, 368
 colorscales, 354
 hover text, 356, 369–370
 marker color, 368
 marker size, 353
 Layout() class, 327
 layout, long format, 367
 x-axis, 368
 y-axis, 368
 gallery, 324
 histogram, 326
 installing, 324
 offline.plot() function, 327
 Python figure reference, 371
 Scattergeo chart type, 352
 user guide, 371
positional arguments. *See* functions:
 arguments
POST requests, 412
printing_models.py, 143–146
Project Gutenberg, 198
.py file extension, 16
Pygame. *See also* Alien Invasion
 project
 background colors, 230–231
 collisions, 268–269, 272–275,
 291–292
 colors, 231
 creating empty windows, 229
 cursor, hiding, 285
 displaying text, 280
 ending games, 276
 event loops, 230
fullscreen mode, 244
groups, 248
 drawing all elements in, 259
 emptying, 271
 looping through, 250
 removing elements from, 251
 storing elements in, 248
 updating all elements in, 249
images, drawing to screen, 235
images, loading, 234
installing, 228
print() calls in, 251
quitting, 244
rect objects
 creating from scratch, 247
 positioning, 234, 247–248,
 257–258, 260–261,
 263–264
responding to input, 230
 keypresses, 238
 mouse clicks, 283
screen coordinates, 234
surfaces, 230
testing games, 270
pyplot module, 307
Python
 >>> prompt, 4
 built-in functions, 471
 documentation, 481
 installing
 on Linux, 470
 on macOS, official, 7–8
 on macOS, using Homebrew,
 469–470
 on Windows, 5–6, 467–469
 interpreter, 16
 keywords, 471
 PEP 8, 68–70, 90, 154
 standard library, 180–181
 terminal sessions, 4
 versions, 4
 Zen of, 30–31
Python Enhancement Proposal
 (PEP), 68
python_repos.py, 361–365
python_repos_visual.py, 366–371

Q

quit values, 118–120

R

random_walk.py, 316–317
random walks, 315–323
 choice() function, 317
 coloring points, 319
 fill_walk() method, 316
 multiple walks, generating, 318
 plotting, 317
 RandomWalk class, 316
 starting and ending points, 320
range() function, 57–59
readlines() method, 188
read() method, 185
refactoring, 206–208, 236
remember_me.py, 204–208
Requests package, 361
return values, 137
r/learnpython, 482
rollercoaster.py, 116
rolling dice, 323–330
 analyzing results, 325
 Die class, 324
 different sizes, 329
 randint() function, 324
 two dice, 328
rubber duck debugging, 480
rw_visual.py, 317–323

S

scatter_squares.py, 311–315
sets, 104
sitka_highs_lows.py, 340–342
sitka_highs.py, 334–340
Slack, 483
sleep() function, 274
slice, 61
split() method, 198
square brackets ([]), 34
squares.py, 58, 60
Stack Overflow, 481
storing data, 202–205. *See also* JSON
strings, 19–25
 changing case, 20
 format() method, 22

f-strings, 21

newlines in, 22

single and double quotes, 19, 24

tabs in, 22

using variables in, 21

whitespace in, 22–24

strptime() method, 338

style guidelines, 68–70

 blank lines, 69

 CamelCase, 181

 classes, 181

 functions, 154

 if statements, 90

 indentation, 69

 line length, 69

 PEP 8, 68

Sublime Text, 4–10, 474–475

 commenting out code, 475

 configuring, 9

 customizing, 474

 indenting and unindenting
 code blocks, 474

 installing, 7–9

 line length indicator, 474

 running Python programs, 9–10

 saving your configuration, 475

 tabs and spaces, 474

subtraction (-), 26

superclasses, 168. *See also* classes:

 inheritance

survey.py, 217

syntax errors, 24

syntax highlighting, 16

T

tab (\t), 22

testing code, 209–222

 adding tests, 214

 assert methods, 216

 coverage, 211

 failing tests, 212–214

 passing tests, 211–212

 setUp() method, 220

 test case, 211

 testing classes, 216–221

 testing functions, 210–215

 unittest module, 209

 unit tests, 211

test_name_function.py, 211–215
test_survey.py, 218–221
text editors and IDEs
 Atom, 476
 Emacs and Vim, 476
 Geany, 476
 IDLE, 475
 Jupyter Notebooks, 477
 PyCharm, 476
 Sublime Text, 4–10, 474–475
 Visual Studio Code, 476
toppings.py, 74, 83–88
traceback, 18
try-except blocks, 194–202. *See also* exceptions
tuples, 65–67
type errors, 66

U

`unittest` module, 209
unit tests, 211
Unix time, 366
user_profile.py, 148

V

values() method, 104
variables, 16–19, 28
 constants, 28
 as labels, 18
 multiple assignment, 28
 naming conventions, 17
 values, 16
version control system, 485.
 See also Git
virtual environment (`venv`), 380
voting.py, 79–80

W

weather data, 334–347
web framework, 379
while loops, 118–127
 active flag, 120–121
 break statement, 121
 infinite, 122
 moving items between lists, 124
 quit values, 118–120
whitespace, 22–24
Windows
 Hello World, running, 10
Python
 checking installed version, 5
 installing, 5–6, 467–469
 troubleshooting installation, 11, 467–469
Sublime Text, installing, 7
terminal
 running programs from, 12
 starting a Python session, 6
with statement, 185
word_count.py, 199–201
write_message.py, 191–193
`write()` method, 192

Z

Zen of Python, 30–31
`ZeroDivisionError`, 194

Python Crash Course, 2nd Edition is set in New Baskerville, Futura, Dogma, and The Sans Mono Condensed.

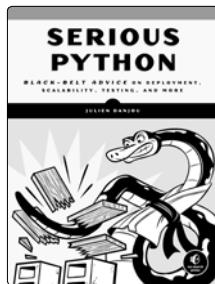
RESOURCES

Visit <https://nostarch.com/pythoncrashcourse2e/> for resources, errata, and more information.

More no-nonsense books from



NO STARCH PRESS

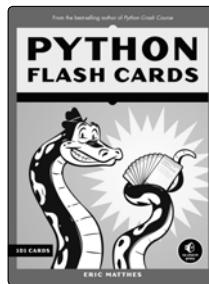


SERIOUS PYTHON

Black-Belt Advice on Deployment, Scalability, Testing, and More

by JULIEN DANJOU

DECEMBER 2018, 240 PP., \$34.95
ISBN 978-1-59327-878-6

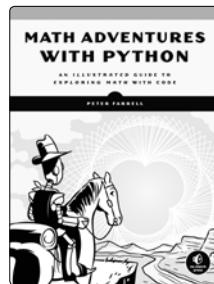


PYTHON FLASH CARDS

Syntax, Concepts, and Examples

by ERIC MATTHES

JANUARY 2019, 101 CARDS, \$27.95
ISBN 978-1-59327-896-0
full color

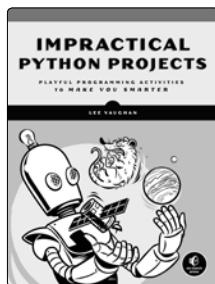


MATH ADVENTURES WITH PYTHON

An Illustrated Guide to Exploring Math with Code

by PETER FARRELL

JANUARY 2019, 304 PP., \$29.95
ISBN 978-1-59327-867-0
full color

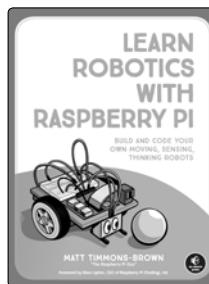


IMPRactical PYTHON PROJECTS

Playful Programming Activities to Make You Smarter

by LEE VAUGHAN

NOVEMBER 2018, 424 PP., \$29.95
ISBN 978-1-59327-890-8

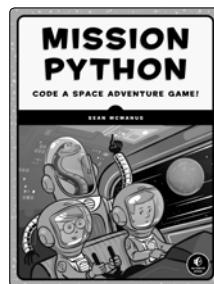


LEARN ROBOTICS WITH RASPBERRY PI

Build and Code Your Own Moving, Sensing, Thinking Robots

by MATT TIMMONS-BROWN

JANUARY 2019, 240 PP., \$24.95
ISBN 978-1-59327-920-2
full color



MISSION PYTHON

Code a Space Adventure Game!

by SEAN McMANUS

OCTOBER 2018, 280 PP., \$29.95
ISBN 978-1-59327-857-1
full color

**WORLDWIDE
BEST SELLER**

**OVER 500,000
COPIES SOLD**



**LEARN PYTHON—
FAST!**

Python Crash Course is the world's best-selling guide to the Python programming language. This fast-paced, thorough introduction to programming with Python will have you writing programs, solving problems, and making things that work in no time.

In the first half of the book, you'll learn basic programming concepts, such as variables, lists, classes, and loops, and practice writing clean code with exercises for each topic. You'll also learn how to make your programs interactive and test your code safely before adding it to a project. In the second half, you'll put your new knowledge into practice with three substantial projects: a *Space Invaders*–inspired arcade game, a set of data visualizations with Python's handy libraries, and a simple web app you can deploy online.

As you work through the book, you'll learn how to:

- Use powerful Python libraries and tools, including Pygame, Matplotlib, Plotly, and Django
- Make 2D games that respond to keypresses and mouse clicks, and that increase in difficulty
- Use data to generate interactive visualizations

- Create and customize web apps and deploy them safely online
- Deal with mistakes and errors so you can solve your own programming problems

This **updated second edition** has been thoroughly revised to reflect the latest in Python code and practices. The first half of the book includes improved coverage of topics like f-strings, constants, and managing data. In the second half, the code for the projects has been updated with better structure, cleaner syntax, and more popular and up-to-date libraries and tools, like Plotly and the latest version of Django. (For a full list of updates, see the Preface.)

If you've been thinking about digging into programming, *Python Crash Course* will get you writing real programs fast. Why wait any longer? Start your engines and code!

ABOUT THE AUTHOR

Eric Matthes is a high school science, math, and programming teacher living in Alaska. He has been writing programs since he was five years old and is the author of the *Python Flash Cards*, also from No Starch Press.

COVERS PYTHON 3.X



THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com

\$39.95 (\$53.95 CDN)

ISBN: 978-1-59327-928-8

5 3 9 9 5



9 781593 279288

SHELF LIFE:
PYTHON
PROGRAMMING LANGUAGE /