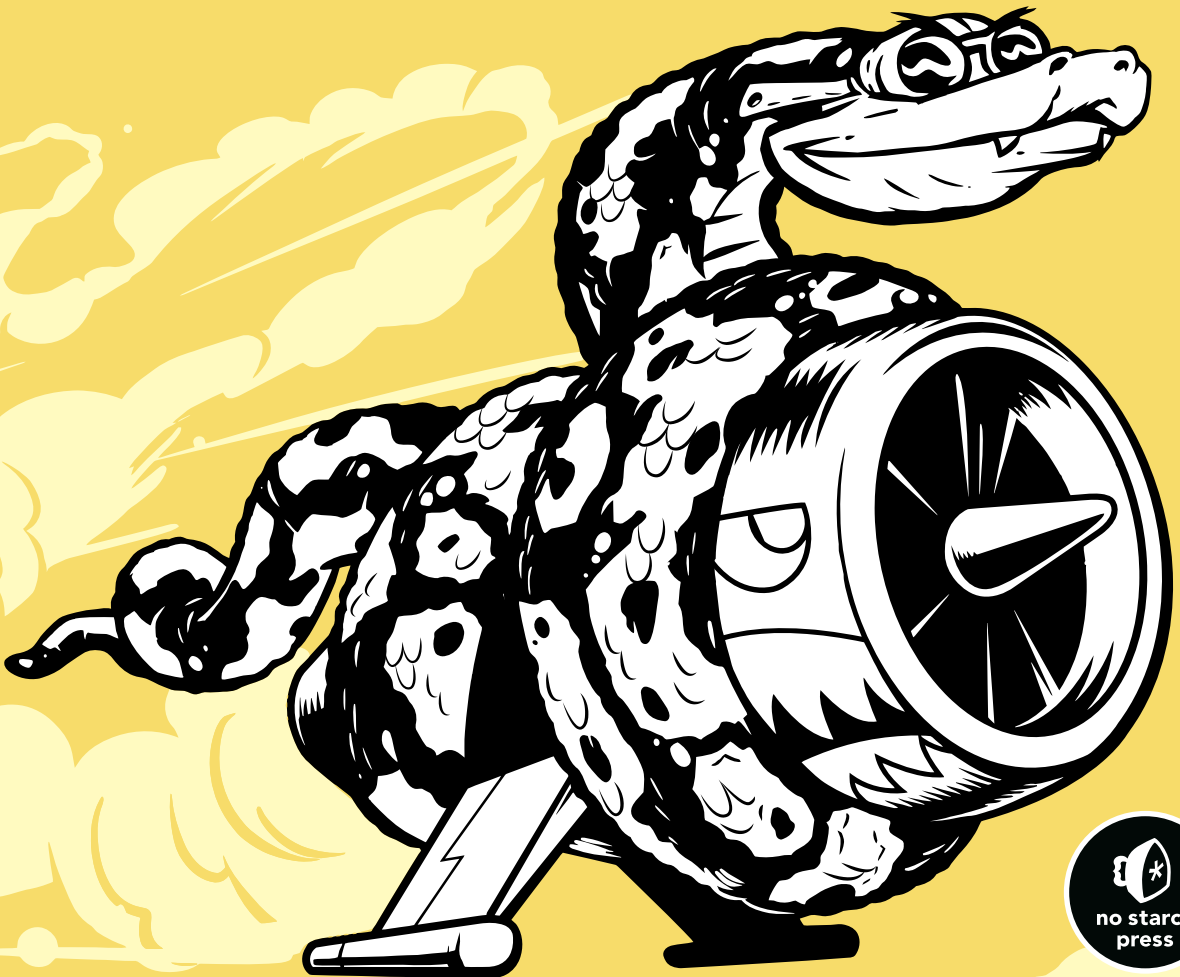


2ND EDITION

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



**no starch
press**

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](https://twitter.com/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 xxxi

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	xxxi
------------------------	-------------

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 hello_world.py	10
Troubleshooting	11
Running Python Programs from a Terminal.	12
On Windows	12
On macOS and Linux	12
<i>Exercise 1-1: python.org</i>	13
<i>Exercise 1-2: Hello World Typos</i>	13
<i>Exercise 1-3: Infinite Skills</i>	13
Summary	13

2 **VARIABLES AND SIMPLE DATA TYPES** **15**

What Really Happens When You Run hello_world.py.	15
Variables	16
Naming and Using Variables	17
Avoiding Name Errors When Using Variables	17
Variables Are Labels	18
<i>Exercise 2-1: Simple Message</i>	19
<i>Exercise 2-2: Simple Messages</i>	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 sort() Method	43
Sorting a List Temporarily with the sorted() 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
<i>Exercise 3-11: Intentional Error</i>	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
<i>Exercise 4-1: Pizzas</i>	56
<i>Exercise 4-2: Animals</i>	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
<i>Exercise 4-3: Counting to Twenty</i>	60
<i>Exercise 4-4: One Million</i>	60
<i>Exercise 4-5: Summing a Million</i>	60
<i>Exercise 4-6: Odd Numbers</i>	60
<i>Exercise 4-7: Threes</i>	60
<i>Exercise 4-8: Cubes</i>	60
<i>Exercise 4-9: Cube Comprehension</i>	60
Working with Part of a List	61
Slicing a List	61
Looping Through a Slice	62
Copying a List	63
<i>Exercise 4-10: Slices</i>	65
<i>Exercise 4-11: My Pizzas, Your Pizzas</i>	65
<i>Exercise 4-12: More Loops</i>	65
Tuples	65
Defining a Tuple	66
Looping Through All Values in a Tuple	67
Writing over a Tuple	67
<i>Exercise 4-13: Buffet</i>	68
Styling Your Code	68
The Style Guide	68
Indentation	69
Line Length	69
Blank Lines	69
Other Style Guidelines	70
<i>Exercise 4-14: PEP 8</i>	70
<i>Exercise 4-15: Code Review</i>	70
Summary	70

5	IF STATEMENTS	71
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
 6	 DICTIONARIES	 91
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 input() Function Works	114
Writing Clear Prompts	114
Using int() 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 break to Exit a Loop	121
Using continue 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

8

FUNCTIONS

129

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

9	CLASSES	157
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

10 FILES AND EXCEPTIONS 183

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
Exercise 10-1: Learning Python	191
Exercise 10-2: Learning C	191
Writing to a File.	191
Writing to an Empty File	191
Writing Multiple Lines	192
Appending to a File.	193
Exercise 10-3: Guest	193
Exercise 10-4: Guest Book	193
Exercise 10-5: Programming Poll.	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
Exercise 10-6: Addition	201
Exercise 10-7: Addition Calculator	202
Exercise 10-8: Cats and Dogs	202
Exercise 10-9: Silent Cats and Dogs	202
Exercise 10-10: Common Words	202
Storing Data	202
Using json.dump() and json.load().	203
Saving and Reading User-Generated Data	204
Refactoring.	206
Exercise 10-11: Favorite Number	208
Exercise 10-12: Favorite Number Remembered	208
Exercise 10-13: Verify User	208
Summary	208

11

TESTING YOUR CODE	209
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
Exercise 11-1: City, Country	215
Exercise 11-2: Population	216
Testing a Class	216
A Variety of Assert Methods	216
A Class to Test	217
Testing the AnonymousSurvey Class	218
The setUp() Method	220
Exercise 11-3: Employee	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
alien_invasion.py	245
settings.py	246
ship.py	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

<i>Exercise 14-3: Challenging Target Practice</i>	288
<i>Exercise 14-4: Difficulty Levels</i>	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
<i>Exercise 14-5: All-Time High Score</i>	301
<i>Exercise 14-6: Refactoring</i>	301
<i>Exercise 14-7: Expanding the Game</i>	302
<i>Exercise 14-8: Sideways Shooter, Final Version</i>	302
Summary	302

PROJECT 2: DATA VISUALIZATION

15	
GENERATING DATA	305
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 <code>scatter()</code>	310
Plotting a Series of Points with <code>scatter()</code>	312
Calculating Data Automatically	312
Defining Custom Colors	314
Using a Colormap	314
Saving Your Plots Automatically	315
<i>Exercise 15-1: Cubes</i>	315
<i>Exercise 15-2: Colored Cubes</i>	315
Random Walks	315
Creating the <code>RandomWalk()</code> Class	316
Choosing Directions	316
Plotting the Random Walk	317
Generating Multiple Random Walks	318
Styling the Walk	319
<i>Exercise 15-3: Molecular Motion</i>	323
<i>Exercise 15-4: Modified Random Walks</i>	323
<i>Exercise 15-5: Refactoring</i>	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
<i>Exercise 15-6: Two D8s</i>	331
<i>Exercise 15-7: Three Dice</i>	331
<i>Exercise 15-8: Multiplication</i>	331
<i>Exercise 15-9: Die Comprehensions</i>	331
<i>Exercise 15-10: Practicing with Both Libraries</i>	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 datetime 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
<i>Exercise 16-1: Sitka Rainfall</i>	346
<i>Exercise 16-2: Sitka–Death Valley Comparison</i>	346
<i>Exercise 16-3: San Francisco</i>	346
<i>Exercise 16-4: Automatic Indexes</i>	347
<i>Exercise 16-5: Explore</i>	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
<i>Exercise 16-6: Refactoring</i>	357
<i>Exercise 16-7: Automated Title</i>	357
<i>Exercise 16-8: Recent Earthquakes</i>	358
<i>Exercise 16-9: World Fires</i>	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
<i>Exercise 17-1: Other Languages</i>		375
<i>Exercise 17-2: Active Discussions</i>		375
<i>Exercise 17-3: Testing python_repos.py</i>		375
<i>Exercise 17-4: Further Exploration</i>		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
<i>Exercise 18-1: New Projects</i>		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
<i>Exercise 18-2: Short Entries</i>		394
<i>Exercise 18-3: The Django API</i>		394
<i>Exercise 18-4: Pizzeria</i>		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>	<i>462</i>
<i>Exercise 20-4: More 404s</i>	<i>462</i>
<i>Exercise 20-5: Extended Learning Log</i>	<i>462</i>
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

Atom	476
Visual Studio Code	476
PyCharm	476
Jupyter Notebooks	477

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.

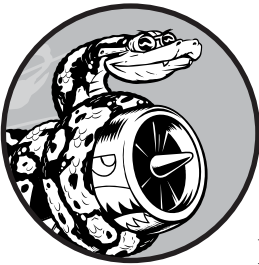
ACKNOWLEDGMENTS

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(message)
 - ❸ NameError: name 'message' 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, *message*, 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:

```
message = "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
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

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`:

```
❶ bicycles = ['trek', 'cannondale', 'redline', 'specialized']
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!"  
❶ print(message)
```

We don't need to indent the `print()` call ❶, 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")  
❶ print("Thank you everyone, that was a great magic show!")
```

Because the line at ❶ 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:

```
❶ ['pizza', 'falafel', 'carrot cake', 'cannoli']
```

My friend's favorite foods are:

```
❷ ['pizza', 'falafel', 'carrot cake', 'ice cream']
```

The output at ❶ shows that 'cannoli' now appears in our list of favorite foods but 'ice cream' doesn't. At ❷ 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:

```
❶ 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 ❶, 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.

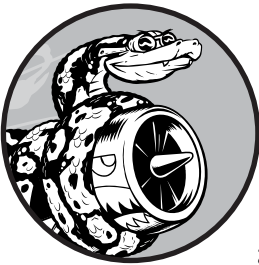
Summary

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    answer = 17
_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:
    price = 0
❶ elif age < 18:
    price = 25
❷ else:
    price = 40
❸

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:
    ❶ if requested_topping == 'green peppers':
        print("Sorry, we are out of green peppers right now.")
    ❷ 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:  
        print(f"Adding {requested_topping}.")  
❺     else:  
        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 ❺. 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.

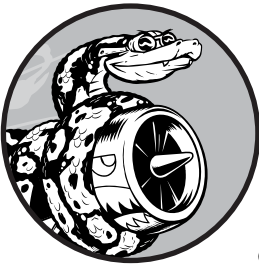
Summary

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 ❺ 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}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'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 ❶ favorite_languages = {
_languages.py    '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 `'aeinstein'` 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:"
            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:"
prompt += "\n(Enter 'quit' when you are finished.) "
```

```
❶ while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        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:
(Enter 'quit' when you are finished.) New York
I'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:
                ❶  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      # Start with users that need to be verified,
_users.py      # 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):

- ❶ File "pets.py", line 6, in <module>
 - ❷ describe_pet()
 - ❸ TypeError: describe_pet() missing 2 required positional arguments: 'animal_type' and 'pet_name'
-

At ❶ 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 ❷ the offending function call is written out for us to see. At ❸ 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 trace-back 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 ❶ def get_formatted_name(first_name, last_name):  
_name.py    """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  
  
    musician = build_person('jimi', 'hendrix')  
    ❸ 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.

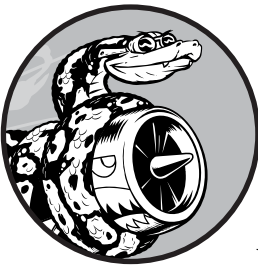
Using functions makes your programs easier to read, and good function names summarize what each part of a program does. Reading a series of function calls gives you a much quicker sense of what a program does than reading a long series of code blocks.

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):
            """Initialize name and age attributes."""
        ❹         self.name = name
            self.age = age

        ❺     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."""

 ❶ 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
❷ else:
 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 ❹ 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)
❶ self.battery_size = 75

❷ 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 class 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 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 filename = 'programming.txt'
_message.py

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:
 answer = int(first_number) / int(second_number)
❷ except ZeroDivisionError:
 print("You can't divide by 0!")
❸ else:
 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 ❸.

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 `FileNotFoundError`: 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 `FileNotFoundError`, 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 `FileNotFoundError` 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:
 ❶ 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 `FileNotFoundError` 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 215830 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 `FileNotFoundError`, 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 import json
_me.py 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 `FileNotFoundError` 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():
 ❶ """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():
 ❶ """Get stored username if available."""
 filename = 'username.json'
 try:
 with open(filename) as f:
 username = json.load(f)
 except FileNotFoundError:
 ❷ 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.

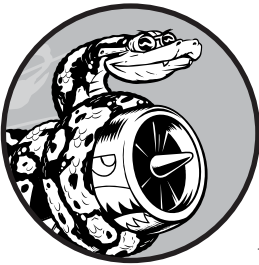
## Summary

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 import unittest
_function.py 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 <i>item</i> is in <i>list</i>        |
| <code>assertNotIn(item, list)</code> | Verify that <i>item</i> is not in <i>list</i>    |

## 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?"
 ❶ self.my_survey = AnonymousSurvey(question)
 ❷ 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.