



Apress®

Learn to

Program with

Python 3

A Step-by-Step Guide to Programming

—

Second Edition

—

Irv Kalb

Learn to Program

with Python 3

A Step-by-Step Guide

to Programming

Second Edition

Irv Kalb

Learn to Program with Python 3

Irv Kalb

Mountain View, California, USA

ISBN-13 (pbk): 978-1-48423878-3

ISBN-13 (electronic): 978-1-4842-3879-0

<https://doi.org/10.1007/978-1-4842-3879-0>

Library of Congress Control Number: 2018954633

Copyright © 2018 by Irv Kalb

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or

information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr

Acquisitions Editor: Todd Green

Development Editor: James Markham

Coordinating Editor: Jill Balzano

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/

rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484238783. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to the memory of my mother, Lorraine Kalb.

*I started learning about programming when I was 16 years old,
at Columbia High School in Maplewood, New Jersey.*

*We were extremely fortunate to have a very early computer,
an IBM 1130, that students could use.*

*I remember learning the basics of the Fortran programming language
and writing a simple program that would add two numbers together
and print the result. I was thrilled when I finally got my program to
work correctly. It was a rewarding feeling to be able to get this huge,
complicated machine to do exactly what I wanted it to do.*

*I clearly remember explaining to my mother that I wrote this
program that got the computer to add 9 and 5 and come up with an
answer of 14. She said that she didn't need a computer to do that.*

I tried to explain to her that getting the answer of 14 was not the

important part. What was important was that I had written a program that would add any two numbers and print the result. She still didn't get it, but she was happy for me and very supportive. Hopefully, through my explanations in this book, you will get it.

Table of Contents

About the Author



xiii

About the Technical Reviewer



Acknowledgments



Chapter 1: Getting Started



1

What Is Python?



2

Installing Python



2

IDLE and the Python Shell



3

Hello World




4

Creating, Saving, and Running a Python

File 
6

IDLE on Multiple Platforms



8

Summary 
9

Chapter 2: Variables and Assignment Statements


11


A Sample Python Program


12


The Building Blocks of Programming


14


Four Types of Data


15

Integers


15

Floats


15

Strings


16

Booleans

16

Examples of Data

17

Form with Underlying Data

18

Variables

19

Assignment Statements

24

v

Table of ConTenTs

Variable Names

27

Naming Convention

28

Keywords

29

Case Sensitivity

31

More Complicated Assignment Statements



33

35

38

39

41

43

43

44

44

45

Errors

46

Syntax Error

46

Exception Error

48

Logic Error

49

Summary

Chapter 3: Built-in Functions

51

Overview of Built-in Functions

51

Function Call

52

Arguments

Results

53

Built-in type Function



53

Getting Input from the User



55

Conversion Functions



56

int Function



57

float Function



57

str Function



57

vi

Table of ConTenTs

First Real Programs



58

Concatenation



61

Another Programming Exercise



62

Using Function Calls Inside Assignment Statements



63

Summary



65

Chapter 4: User-Defined Functions



67

A Recipe as an Analogy for Building Software



68

Ingredients



68

Directions



68

Definition of a Function



71

Building Our First Function



72

Calling a User-Defined Function



73

Receiving Data in a User-Defined Function: Parameters



76

Building User-Defined Functions with Parameters

78

Building a Simple Function That Does Addition

81

Building a Function to Calculate an Average

81

Returning a Value from a Function: The return Statement

82

Returning No Value: None

84

Returning More Than One Value

85

Specific and General Variable Names in Calls and Functions

86

Temperature Conversion Functions

88

Placement of Functions in a Python File

89

Never Write Multiple Copies of the Same Code

90

Constants

91

Scope

93

Global Variables and Local Variables with the Same Names

97

Finding Errors in Functions: Traceback

98

Summary
101

vii

Table of ConTenTs

Chapter 5: if, else, and elif Statements

103

Flowcharting

104

The if Statement

107

Comparison Operators

109

Examples of if Statements

109

Nested if Statement



111

The else Statement



111

Using if/else Inside a Function



114

The elif Statement



115

Using Many elif Statements



118

A Grading Program



120

A Small Sample Program: Absolute Value



120

Programming Challenges



123

Negative, Positive, Zero



123

isSquare



125

isEven



128

isRectangle



130

Conditional Logic



132

The Logical not Operator



132

The Logical and Operator



133

The Logical or Operator



135

Precedence of Comparison and Logical Operators



136

Booleans in if Statements



136

Program to Calculate Shipping



137

Summary



141

Chapter 6: Loops



143

User's View of the Game



144

Loops



145

The while Statement



147

First Loop in a Real Program



150

viii

Table of ConTenTs

Increment and Decrement



151

Running a Program Multiple Times



152

Python's Built-in Packages



154

Generating a Random Number



155

Simulation of Flipping a Coin



157

Other Examples of Using Random Numbers



158

Creating an Infinite Loop



160

A New Style of Building a Loop: while True, and break



160

Asking If the User Wants to Repeat: the Empty String



163

Pseudocode



164

Building the Guess the Number Program



164

Playing a Game Multiple Times



171

Error Detection with try/except



173

The continue Statement



175

Full Game


176

Building Error-Checking Utility Functions


178

Coding Challenge



179

Summary 
181


Chapter 7:

Lists 
183


Collections of Data


184

Lists


185

Elements


185

Python Syntax for a List


186

Empty List


187

Position of an Element in a List: Index



187

Accessing an Element in a List



189

Using a Variable or Expression as an Index in a List



190

Changing a Value in a List



192

Using Negative Indices



192

Building a Simple Mad Libs Game



193

ix

Table of ConTenTs

Adding a List to Our Mad Libs Game



195

Determining the Number of Elements in a List: The len Function



196

Programming Challenge 1



198

Using a List Argument with a Function



[200](#)

[Accessing All Elements of a List: Iteration](#)



[203](#)

[for Statements and for Loops](#)



[204](#)

[Programming Challenge 2](#)



[206](#)

[Generating a Range of Numbers](#)



[207](#)

[Programming Challenge 3](#)



[208](#)

[Scientific Simulations](#)



[209](#)

[List Manipulation](#)



[214](#)

[List Manipulation Example: an Inventory](#)



[216](#)

[Pizza Toppings Example](#)



[217](#)

[Summary](#)



[223](#)

Chapter 8: Strings



225

len Function Applied to Strings



226

Indexing Characters in a String



226

Accessing Characters in a String



227

Iterating Through Characters in a String



228

Creating a Substring: A Slice



230

Programming Challenge 1: Creating a Slice



232

Additional Slicing Syntax



235

Slicing as Applied to a List



236

Strings Are Not Changeable



236

Programming Challenge 2: Searching a String



237

Built-in String Operations



238

Examples of String Operations



240

Programming Challenge 3: Directory Style



241

Summary



243

x

Table of ConTenTs

Chapter 9: File Input/Output



245

Saving Files on a Computer



246

Defining a Path to a File




247

Reading from and Writing to a File



249

File Handle


250

The Python os Package


251


Building Reusable File I/O Functions


252

Example Using Our File I/O Functions


254


Importing Our Own Modules


255


Saving Data to a File and Reading It Back


257


Building an Adding Game


260

Programming Challenge 1


260

Programming Challenge 2


261

Writing/Reading One Piece of Data to and from a File


263

Writing/Reading Multiple Pieces of Data to and from a File


266

The join Function


266

The split Function


267

Final Version of the Adding Game


268


Writing and Reading a Line at a Time with a File


271

Example: Multiple Choice Test



275

A Compiled Version of a Module



281

Summary 
282

Chapter 10: Internet Data


283

Request/Response Model


283

Getting a Stock Price



285

Pretending to Be a Browser



286

API



288

Requests with Values



288

API Key



289

Example Program to Get Stock Price Information Using an API



291

xi

Table of ConTenTs

Example Program to Get Weather Information



294

URL Encoding



297

Summary



299

Chapter 11: Data Structures



301

Tuples



302

Lists of Lists



305

Representing a Grid or a Spreadsheet



306

Representing the World of an Adventure Game



307

Reading a Comma-Separated Value (CSV) File



311

Dictionary



316

Using the in Operator on a Dictionary



319

Programming Challenge



320

A Python Dictionary to Represent a Programming Dictionary



322

Iterating Through a Dictionary



323

[Combining Lists and Dictionaries](#)



[325](#)

[JSON: JavaScript Object Notation](#)



[328](#)

[Example Program to Get Weather Data](#)



[331](#)

[XML Data](#)



[334](#)

[Accessing Repeating Groupings in JSON and XML](#)



[338](#)

[Summary](#)



[341](#)

[Chapter 12: Where to Go from Here](#)



[343](#)

[Python Language Documentation](#)



[343](#)

[Python Standard Library](#)



[344](#)

[Python External](#)



[Packages](#)

[345](#)

[Python Development Environments](#)

346

Projects and Practice, Practice, Practice

Summary

[Index](#)

xii



Irv Kalb is an adjunct professor at UCSC (University of California, Santa Cruz) Extension Silicon Valley and Cogswell Polytechnical College. He has been teaching software development classes since 2010

software development classes since 2010.

Irv has worked as a software developer, manager of software developers, and manager of software development projects. He has been an independent consultant for many years with his own company, Furry Pants Productions, where he has concentrated on educational software. Prior to that, he worked as an employee for a number of high-tech companies. He has BS and MS degrees in computer science.

Recently, he has been a mentor to a number of local competitive robotics teams.

His previous publications include numerous technical articles, two children's edutainment CD-ROMs (about Darby the Dalmatian), an online e-book on object-oriented programming in the Lingo programming language, and the first book on Ultimate Frisbee, *Ultimate: Fundamentals of the Sport* (Revolutionary Publications, 1983).

He was highly involved in the early development of the sport of Ultimate Frisbee.

xiii

About the Technical Reviewer

Mark Furman, MBA is a systems engineer, author, teacher, and entrepreneur. For the last 16 years he has worked in the information technology field with a focus on Linux-based systems and programming in Python. He's worked for a range of companies

including Host Gator, Interland, Suntrust Bank, AT&T, and Winn-Dixie. Currently he has been focusing his career on the maker movement and has launched Tech Forge

(techforge.org), which focuses on helping people start a makerspace and help sustain current spaces. He holds an MBA degree from Ohio University. You can follow him on Twitter @mfurman.

xv

Acknowledgments

I would like to thank the following people, without whom this book would not have been possible:

My wonderful wife, Doreen, who is the glue that keeps our family together.

Our two sons, Jamie and Robbie, who keep us on our toes.

Our two cats, Chester and Cody (whom we think of as people).

Mariah Armstrong, who created all the graphics in this book. I am not an artist

(I don't even play one on TV). Mariah was able to take my "chicken scratches" and turn them into very clear and understandable pieces of art.

Chris Sasso and Ravi Chityala for their technical reviews and helpful suggestions.

Luke Kwan, Catherine Chanse, and Christina Ri at the Art Institute of California-Silicon Valley.

Andy Hou at the UCSC-Silicon Valley Extension.

Jerome Solomon at Cogswell Polytechnical College, who first suggested that I consider getting into Python.

Jill Balzano, Jim Markham, Mark Furman, and Todd Green at Apress for all the work they did reviewing, editing, and expertly answering all my questions.

All the students who have been in my classes over many years at the Art Institute California-Silicon Valley, Cogswell Polytechnical College, and the UCSC Silicon Valley Extension. Their feedback, suggestions, smiles, frowns, light-bulb moments, frustrations, and knowing head-nods were extremely helpful

in shaping the content of this book.

Finally, Guido van Rossum, without whom Python would not exist.

xvii

CHAPTER 1

Getting Started

Congratulations! You have made a wise decision. No, not the decision to buy this book, although I think that will turn out to be a wise decision also. I mean you have made a wise decision to learn the basics of computer programming using the Python language.

In this book, I teach you the fundamentals of writing computer software. I assume that you have never written any software before, so I start completely from scratch. The only requirements are that you possess a basic knowledge of algebra and a good sense of logic. As the book progresses, each chapter builds upon the information learned in the previous chapter(s). The overall goal is to give you a solid introduction to the way that computer code and data interact to form well-written programs. I introduce the key elements of software, including variables, functions, if/else statements, loops, lists, and strings. I offer many real-world examples that should help explain the uses of each of these elements. I also give definitions to help you with the new vocabulary that I introduce.

This book is not intended to be comprehensive. Rather, it is an introduction that gives you a solid foundation in programming. The approach is highly interactive, asking you to create small programs along the way as a chance to practice what has been explained in each chapter. By the end of the book, you should be comfortable writing small to medium-sized programs in Python.

This first chapter covers the following topics:

- Introducing Python
- Getting Python installed on your computer
- Using IDLE and the Python Shell
- Writing your first program: Hello World

- Creating, saving, and running Python files
- Working with IDLE on multiple platforms

1

© Irv Kalb 2018

I. Kalb, *Learn to Program with Python 3*, https://doi.org/10.1007/978-1-4842-3879-0_1

Chapter 1 GettinG Started

What Is Python?

Python is a general-purpose programming language. That means it was designed and developed to write software for a wide variety of disciplines. Python has been used to write applications to solve problems in biology, chemistry, financial analysis, numerical analysis, robotics, and many other fields. It is also widely used as a *scripting language* for use by computer administrators, who use it to capture and replay sequences of computer commands. It is different from a language like HTML (HyperText Markup Language), which was designed for the single purpose of allowing people to specify the layout of a web page.

Once you learn the basic concepts of a programming language like Python, you find that you can pick up a new computer languages very quickly. No matter what the language (and there are many) the underlying concepts are very similar. The key things that you learn about—variables, assignment statements, if statements, while loops, function calls—are all concepts that are easily transferable to any other programming language.

Installing Python

Python was created in the 1990s by Guido van Rossum. He is affectionately known as Python’s Benevolent Dictator for Life. The language has two current versions: 2.7 and 3.6. Version 2.7 is still widely used, but its “end of life” has recently been announced.

Therefore, this version of the book will use the newer *Python 3*, as it is known. With respect to the contents of this book, there are only a few differences between the versions of the language. Where appropriate, I point out how something presented in Python 3

was handled in Python 2.

Python is maintained as an *open source* project by a group called the Python Software Foundation. Because it is open source, Python is free. There is no single company that owns and/or sells the software. You can get everything you need to write and run all the Python programs in this book by simply downloading Python from the Internet. I'll explain how you can get it and install it.

The center of the Python universe is at www.python.org.

Bring up the browser of your choice and go to that address. The site changes over time, but the essential functionality should remain the same. On the main page, there should be a Downloads button or rollover. Once you're in the Downloads area, you 2

Chapter 1 GettinG Started

should be able to select Windows, Mac, or Other Platforms (which includes Linux).

After choosing your operating system, you should get an opportunity to choose between versions 3.x.y (whatever is the current subversion of Python 3) and version 2.x.y (whatever is the current subversion of Python 2). Choose version 3.x.y.

Clicking the button downloads an installer file. On a Mac, the downloaded file has a name like python-3.6.4-macosx10.6pkg. On a Windows computer, the file has a

name like python-3.6.4-msi. On either platform, find the file that was downloaded and double-click it. That should start the installation process, which should be very simple.

IDLE and the Python Shell

There are many different *software development environments* (applications) that you can use to write code in Python. It may seem odd that you use a program to write a program, but that's what a software development environment is. Some of these environments are free; others can be costly. They differ in the tools they offer to help programmers be more efficient.

The environment we will use in this book is called IDLE. You might think that IDLE

is an acronym, maybe Interactive DeveLopment Environment. When the name was

chosen, it didn't mean anything. In fact, the name Python doesn't refer to the snake.

Apparently, Guido van Rossum was a big fan of *Monty Python's Flying Circus*, a TV

series by a well-known comedy group from Britain, and he named the language after them. One of the founding members of Monty Python was Eric Idle. The name IDLE is a reference to him.

IDLE is free. When you download and run the Python installer, it installs IDLE on your computer. Once installed, you can find IDLE on a Mac by opening the applications folder and locating the folder named Python 3.x. Once you open it, you should see the IDLE application. To open IDLE, double-click the icon. On Windows, IDLE is installed in the standard Program Files folder. If your version of Windows has a Start button, click the Start button and type **IDLE** in the type-in field. Otherwise, you might have to do a Control+R or Control+Q to bring up a dialog box where you can type **IDLE**. However you open IDLE, you should see a window with contents that look something like this:

```
Python 3.6.1 (v3.6.1:69c0db5050, Mar 21 2017, 01:21:04)
```

```
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
```

```
Type "copyright", "credits" or "license()" for more information.
```

```
>>>
```

```
2
```

Chapter 1 GettinG Started

This window is called the Python Shell. In fact, the title of the window should be Python 3.x.y Shell.

Hello World

There is a tradition that when programmers learn a new computer language, they try writing what is called the Hello World program. That is, just to make sure they can get something to work, they write a simple program that writes out “Hello World!”

Let’s do that now with Python. The Python Shell (commonly just called the Shell) gives you a prompt that looks like three greater-than signs. This is called the *chevron prompt* or simply the *prompt*. When you see the prompt, it means the Shell is ready for you to type something. Throughout this book, I strongly encourage you to use the IDLE

environment by trying out code as I explain it. At the prompt, enter the following:

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

Then press the Return key or Enter key. When you do, you should see this:

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

```
Hello World!
```

```
>>>
```

Congratulations! You have just written your first computer program. You told the computer to do something, and it did exactly what you told it to do. My work is done here. You’re not quite ready to add *Python programmer* to your résumé and get a job as a professional computer programmer, but you are off to a good start!

Note if you don’t like the font and/or size of the text used in the Shell, you can choose idLe ► preferences (Mac) or Configure idLe (Windows) and easily

change

either or both.

One of the key advantages of the Python language is how readable it is. The program you just wrote is simply the word `print`, an open parenthesis, whatever you want to be printed (inside quotes), and a closing parenthesis. Anyone can understand the Hello World program written in Python. But to make this point very clear, let's see what you have to do to write the Hello World program in some other popular languages.

4

Chapter 1 GettinG Started

You've probably heard of the language called C, perhaps the most widely used programming language in the world. Here is what you have to write in C to get the same results:

```
#include <stdio.h>

int main(void)

{

printf("Hello World!\n");

return 0;

}
```

Notice all the brackets, parentheses, braces, and semicolons you need to have, along with how many lines you have to write?

There is another language called C++, which is a modification of the original C language to give it more power. Here's what the Hello World program looks like in C++:

```
#include "std_lib_facilities.h"
```

```

int main()

{

cout << "Hello World!\n";

return 0;

}

```

Not surprisingly, it also has many brackets, parentheses, braces, and semicolons.

Finally, here is the same Hello World program written in Java, yet another popular computer language:

```

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello World!");

}

}

```

Again, there are many brackets, parentheses, and semicolons, and many words with meanings that are not immediately obvious.

By comparison, notice how English-like, simple, and readable the Python version

is. This readability and simplicity are big reasons why Python is growing in popularity, especially as a language used to teach programming to beginners.

5

Chapter 1 GettinG Started

Creating, Saving, and Running a Python File

So far, you have only seen a single line of Python code:

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

You typed it into the Shell and pressed Enter or Return to make it run. Typing one line at a time into the Shell is a great way to learn Python, and it is very handy for trying out things quickly. But soon I'll have you writing programs with tens, hundreds, and maybe thousands of lines of code. The Shell is not an appropriate place for writing large programs. Python, like every other computer language, allows you to put the code you write into a file and save it. Programs saved this way can be opened at any time and run without having to retype them. I'll explain how we do this in Python.

Just like any standard word processor or spreadsheet program, to create a new file in IDLE, you go to the File menu and select New File (denoted from here on as File ► New File). You can also use the keyboard shortcuts Control+N (Windows) or Command+N

(Mac).

This opens a new, blank editing window, waiting for you to enter Python code. It behaves just like any text editing program you have ever used. You enter your Python code, line by line, similar to the way that you did it in the Shell. However, when you press Return or Enter at the end of a line, the line does not run—it does *not* produce immediate results as it did in the Shell. Instead, the cursor just moves down to allow you to enter another line. You can use all the standard text-editing features that you are used to: Cut, Copy, Paste, Find, Replace, and so on. You can move around the lines of code using the arrow keys or by clicking the mouse. When a program gets long enough, scrolling becomes enabled. You can select multiple lines using the

standard click-and-drag or click to create a starting point and Shift-click to mark an ending point.

Let's build a simple program containing three print statements. Open a new file.

Notice that when you open the file, it is named Untitled in the window title. Enter the following:

```
print('I am now entering Python code into a Python file.')
```

```
print('When I press Return or Enter here, nothing happens.')
```

```
print('This is the last line.')
```

6

Chapter 1 GettinG Started

When you type the word **print**, IDLE colorizes it (both here in the editing window and when you type it in the Shell). This is IDLE letting you know that this is a word that it recognizes. IDLE also turns all the words enclosed in quotes to green. This also is an acknowledgement from IDLE that it has an understanding of what you are trying to say.

Notice that when you started typing, the window title changed to *Untitled*. The asterisks around the name are there to show that the contents of the file have been changed, but the file has not been saved. That is, IDLE knows about the new content, but the content has not yet been written to the hard disk. To save the file, press the standard Control+S (Windows) or Command+S (Mac). Alternatively, you can click File ► Save.

Because this is the first time the file is being saved, you see the standard Save dialog box.

Feel free to navigate to a folder where you are able to find your Python files(s), or click the New Folder button to create a new folder. In the top of the box, where it says “Save As”, enter a name for this file. Because we are just testing things out, you can name the file Test. However, Python filenames should always end with a .py extension. Therefore, you should enter the name **Test.py** in the Save As box.

Note if you save your python file without a .py extension, idLe will not recognize it as a python file. if python does not know that your file is a python file, it will not colorize your code. this may not seem important now, but it will turn out to be very helpful when you start writing larger programs. So make it a habit right from the start to always ensure that your python file names end with the.py extension.

Now that we have a saved Python file, we want to run, or *execute*, the *statements* in the file. To do that, click Run ► Run Module or press the F5 shortcut key. If everything went well, the program should print the following in the Shell:

```
>>> print('This is the last line.')
This is the last line.
>>>
```

I am now entering Python code into a Python file.

When I press Return or Enter here, nothing happens.

This is the last line.

Now let's quit IDLE by pressing Control+Q (Windows) or Command+Q (Mac) keys.

Alternatively, you can click IDLE ► Exit (Windows) or IDLE ► Quit IDLE (Mac).

When you are ready to open IDLE again, you have choices. You can open IDLE by

typing **IDLE** into the Start menu (Windows) or by double-clicking the IDLE icon (Mac).

If you then want to open a previously saved Python file, you can click File ► Open and navigate to the file you want to open.

7

Chapter 1 GettinG Started

However, if you want to open IDLE and open a previously saved Python file, you can navigate to the saved Python file (for example, find the Test.py file that you just saved) and open IDLE by opening the file. On Windows, if you double-click the icon, a window typically opens and closes very fast. This runs the Python program, but does not keep the window open. Instead, to open the file and IDLE, right-click the file icon. From the context menu that appears, select the second item, Edit with IDLE.

On a Mac, you can simply double-click the file icon. If double-clicking the Python file opens a program other than IDLE, you can fix that with a one-time change. Quit whatever program opened. Select the Python file. Press Command+I (or click File ►

Get Info), which opens a long dialog box. In the section labeled "Open with", select the IDLE application (IDLE.app). Finally, click the Change All button. Once you do that, you should be able to double-click any file whose name ends

in .py, and it should open with IDLE.

Programming typically involves iterations of edits to one or more Python files.

Each time you make changes and you want to test the new code, you must save the

file and then run it. If you don't save the file before you try to run it, IDLE will prompt you by asking you to save the file. You'll quickly become familiar with the typical development cycle of edit your code, save the file (Command+S or Control+S), and run the program (F5).

IDLE on Multiple Platforms

One other very nice feature of Python and IDLE is that the environment is almost completely platform independent. That is, the IDLE environment looks almost identical on a Windows computer, Mac, or Linux system. The only differences are those associated with the particular operating system (such as the look of the window's title bar, the location of the menus, the look of the dialog boxes, and so on). These are very minor details. Overall, the platform you run on does not matter.

Perhaps even more importantly, the code you write is platform independent. If

you create a Python file on one platform, you can move that file to another platform and it will open and run just fine. Many programmers use multiple systems to develop Python code. In fact, even though I typically develop most of my Python code on a Mac, I often bring these same files into classrooms, open them, teach with them on Windows systems.

8

Chapter 1 GettinG Started

Summary

In this chapter, you got up and running with Python. You should now have Python

installed on your computer and have a good understanding of what the IDLE

environment is. You built the standard Hello World program in the Shell, and then used the editor window to build, save, and run a simple multiline Python program (whose name ends in .py) made up of print statements. Finally, you learned that Python and the IDLE environment are platform independent.

9

CHAPTER 2

Variables and Assignment

Statements

This chapter covers the following topics:

- A sample Python program
- Building blocks of programming
- Four types of data
- What a variable is
- Rules for naming variables
- Giving a variable a value with an assignment statement
- A good way to name variables
- Special Python keywords
- Case sensitivity
- More complicated assignment statements
- Print statements
- Basic math operators
- Order of operations and parentheses
- A few small sample programs

A few small sample programs

- Additional naming conventions
- How to add comments in a program
- Use of “whitespace”
- Errors in programs

11

© Irv Kalb 2018

I. Kalb, *Learn to Program with Python 3*, https://doi.org/10.1007/978-1-4842-3879-0_2

Chapter 2 Variables and assignment statements

A Sample Python Program

Let’s jump right in and see an example of what Python code looks like. You are probably familiar with a simple toy called the Magic 8-Ball, made by Mattel, Inc. To play with the toy, you ask it a yes-or-no question, turn the ball over, and the ball gives you one of a number of possible answers. Here is the output of a Python program that simulates the Magic 8-Ball:

Ask the Magic 8-Ball a question (Return or Enter to quit): Will this be a great book?

Absolutely!

Ask the Magic 8-Ball a question (Return or Enter to quit): Will I learn to program in Python?

Answer is foggy, ask again later.

Ask the Magic 8-Ball a question (Return or Enter to quit): Will I learn to program in Python?

You may rely on it.

Ask the Magic 8-Ball a question (Return or Enter to quit): Will I be able to play football in the NFL?

No way, dude!

Ask the Magic 8-Ball a question (Return or Enter to quit): Will I make a million dollars?

Absolutely!

Ask the Magic 8-Ball a question (Return or Enter to quit): Does the Magic 8-Ball ever make mistakes?

No way, dude!

Ask the Magic 8-Ball a question (Return or Enter to quit):

12

Chapter 2 Variables and assignment statements

Now, let's jump right in and take a look at the underlying code of this program.

I'm showing you this just to give you a feeling for what Python code looks like. I am certainly not expecting you to understand much of this code. At this point, the details are unimportant. Here it is:

```
import random # Allow the program to use random numbers
```

```
while True:
```

```
    print() # prints a blank line
```

```
    usersQuestion = input('Ask the Magic 8-Ball a question
```

```
(press enter to quit): ')
```

```
if usersQuestion == "":
    break # we're done

randomAnswer = random.randrange(0, 8) # pick a random number

if randomAnswer == 0:
    print('It is certain.')
elif randomAnswer == 1:
    print('Absolutely!')
elif randomAnswer == 2:
    print('You may rely on it.')
elif randomAnswer == 3:
    print('Answer is foggy, ask again later.')
elif randomAnswer == 4:
    print('Concentrate and ask again.')
elif randomAnswer == 5:
    print('Unsure at this point, try again.')
elif randomAnswer == 6:
    print('No way, dude!')
elif randomAnswer == 7:
    print('No, no, no, no, no.')
```

Here's a very quick explanation: at the top, there is a line that allows the program to use random numbers. Then there is a line that says `while True`. This line creates something called a *loop*, which is a portion of a program that runs over and over again.

In this case, it allows the user to ask a question and get an answer, and then enter another question and get another answer, and on and on.

Moving down, there is a line that causes Ask the Magic 8-Ball a question to be printed out and allows the user to type a question for the Magic 8-Ball to answer.

Skipping down a few lines, the program generates a random number between 0 and 7.

After generating the random number, the program then checks to see if the value of the random number is 0. If so, it tells the user the answer: It is certain. Otherwise, if the value of the randomly chosen number is 1, it tells the user: You may rely on it.

The rest of the lines work similarly, checking the random number and giving different outputs.

After the program prints an answer, because the program is inside the loop, the program goes around again and tells the user to ask another question. And the process keeps going.

As I said, don't worry about the details of the program—just get a sense of how the program does what it does. But there are some things to notice. First, see how readable this code is. With only this brief introduction, you can probably get a feeling for the basic logical flow of how the program operates. Second, notice that the program asks the user for input, does some computation, and generates some output. These are the three main steps in almost all computer programs.

Let's get into programming 101. This may be extremely basic, but I want to start right at the beginning, create a solid foundation, and then build on that.

The Building Blocks of Programming

The two basic building blocks of programming are code and data. *Code* is a set of instructions that tell the computer what to perform and how it should perform. But I want to start our discussion with data.

Data refers to the quantities, characters, and/or symbols on which operations are performed with a computer. Anything you need the computer to remember is a piece of data. Simple examples of data include the number of students in class, grade point average, name, whether a switch is in an on or off position, and so on.

14

Chapter 2 Variables and assignment statements

There are many different types of data, but this book deals mostly with four basic types, which I describe in the next section.

Four Types of Data

The four basic types of data are called *integer numbers*, *floating-point numbers*, *strings*, and *Booleans*. This section explains and provides examples of each of these types of data.

Integers

Integer numbers (or simply, *integers*) are counting numbers, like 1, 2, 3, but also include 0 and negative numbers. The following are examples of data that is expressed as integers:

- Number of people in a room
- Personal or team score in a game
- Course number
- Date in a month
- Temperature (in terms of number of degrees)

Floats

Floating-point numbers (or simply *floats*) are numbers that have a decimal point in them. The following are examples of data that is expressed as floating-point numbers:

- Grade point average
- Price of something
- Percentages
- Irrational numbers, like pi

15

Chapter 2 Variables and assignment statements

Strings

Strings (also called *text*) are any sequences of characters. Examples of data that is expressed as strings include the following:

- Name
- Address
- Course name
- Title of a book, song, or movie
- Sentence
- Name of a file on a computer

Booleans

Booleans are a type of data that can only have one of two values: True or False. Booleans are named after the English mathematician George Boole, who created an entire field of logic based on these two-state data items. The following are some examples of data that can be expressed as Booleans:

- *The state of a light switch*: True for on, False for off
- *Inside or outside*: True for inside, False for outside
- *Whether someone is alive or not*: True for alive, False for dead
- *If someone is listening*: True for listening, False for not listening

It might seem that integer and floating-point data have overlaps. For example, there is an integer 0 and there is a floating-point 0.0. There is an integer 1 and a floating-point 1.0. Although these may appear to be the same thing to us humans, integers and floats are handled very differently inside the computer. Without getting too wrapped up in the details, it is easier for the computer to represent and operate with integers. But when we have a value with a decimal point, we need to use a floating-point number instead.

Whenever we represent a value, we choose the appropriate numeric data type. As you will see, Python makes a clear distinction between these two types of data.

There are many other types of data in the computer world. For example, you are probably familiar with music being stored in MP3 format or video being stored in MP4.

These are other representations of data. However, to make things simple and clear, I'll use just the four basic types of data in most of this book.

16

Chapter 2 Variables and assignment statements

Examples of Data

Now let's take a look at what the actual data looks like for each of the four different data types.

- **Integer** numbers are whole or counting numbers. These are some examples:

12, 50, 0, -3, -25

- **Floating-point** numbers are any numbers that contain a decimal point. These are some examples:

1.5, .5, -3.21, 1.0, 0.0

- **Strings** represent textual data or any sequence of characters. String data is always represented with quote characters before and after the sequence of characters. In Python, you can use either the single (') or the double-quote character ("). The following are examples of strings:

'Joe', 'Schmoe', "Joe", "Schmoe", 'This is some string data', "OK"

The string 'Joe' and the string "Joe" are exactly the same. The quotes are not actually part of the string. They are there to allow Python to understand that you are talking about a string. You can choose to use either pair of quoting characters. Single quotes are generally easier to use because you don't have to hold down the Shift key to type them.

However, if you want to include a quote character inside a string, you can enclose the string in the other quote characters. For example, you might write this:

"Here's a string with a single quote in it"

Or this:

'Here is a string that includes two "double quote" symbols

inside of it'

Think back to the Hello World program you wrote in [Chapter 1](#).

The 'Hello World!' you used was an example of a string.

17

Widgets'R'Us	
Name:	<input type="text"/>
Address:	<input type="text"/>
Number of Widgets:	<input type="text"/>
Total to Pay:	<input type="text"/>
Receipt?:	<input type="checkbox"/>

Chapter 2 Variables and assignment statements

- **Boolean** data can only have one of two values: True or False. The words *True* and *False* must be spelled with this capitalization:

True

False

Form with Underlying Data

To make the distinctions among these different data types clearer, let's look at a fake but typical form you might see if you bought something online. Imagine you want to buy some widgets (generic items), and you go to the WidgetsRUs.com web site to buy them.

You might be presented with a form like the one shown in Figure [2-1](#).

Figure 2-1. *Sample form where the fields represent different types of data* As the end user, you would type characters into each of these fields. But as the programmer who is writing this program, you have to think about what types of data you would use to represent the information that the user entered into these fields.

The first two fields, Name and Address, call for string data. The user enters characters, and we would think of what they wrote as strings.

The Number of Widgets field represents a piece of integer data—for example, 10. It wouldn't make sense to order 12 and a half widgets, so this is certainly an integer.

18

Chapter 2 Variables and assignment statements

Total to Pay would be a floating-point number, such as 37.25. We think of money

written as dollars, a decimal point, and cents. So you would use a floating-point piece of data to represent this.

Receipt is what an end user commonly sees as a check box. But if you were writing the program behind this form, you would represent the answer to the receipt question with a Boolean: True if the box is checked on, False if the box is unchecked. From now on, no matter what device you see a form like this on, you will see check boxes differently. Every time you see a check box, you'll realize that the underlying program is representing your choice with a piece of Boolean data.

Variables

In programming, we need to remember and manipulate data all the time. This is a

fundamental part of computer programming. In order to store and manipulate data, we use a variable.

Definition a

variable is a named memory location that holds a value.

The contents of a variable can change or vary over time; this is why it's called a *variable*.

You've probably heard that the term RAM stands for *random-access memory*. It is the active part of storage inside your computer. You can think of this memory as a simple list or array of numbered slots, starting at 0 and going up to as much memory as you have in your computer. The amount doesn't matter, but Figure [2-2](#) is a diagram showing memory starting a slot 0 and going up to the final slot of 4 gigabytes.

--

--

--

--

THE

WORLD

OF

THE

FUTURE

--

--

--

--

Chapter 2 Variables and assignment statements

Figure 2-2. Random-access memory diagram

Every one of these memory locations can be used as a variable. That is, you can store a piece of data in any free memory slot.

20

Chapter 2 Variables and assignment statements

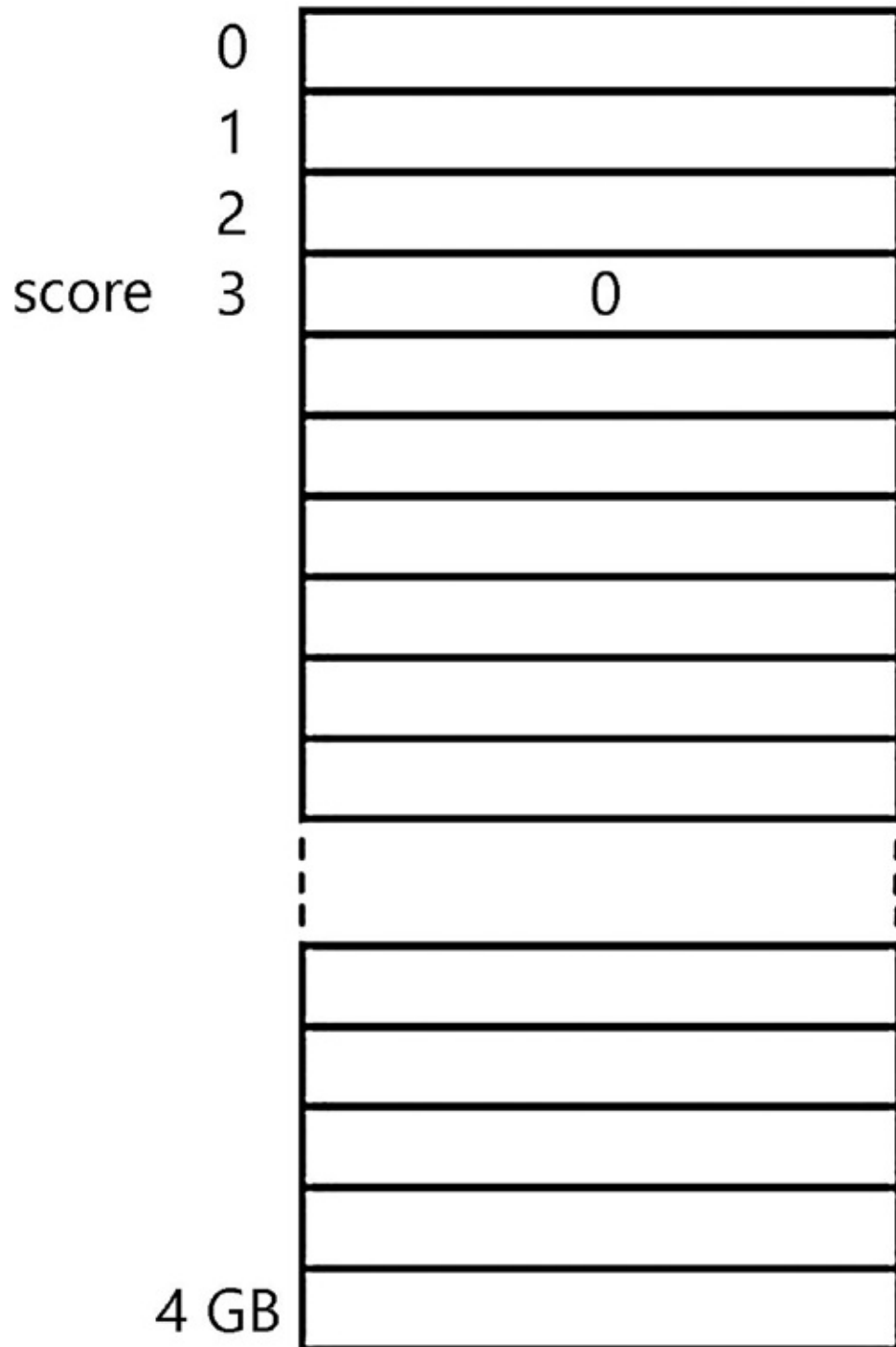
Note behind the scenes, the way python stores data and variable names is more complex. For example, different types of data (integer, float, string, and boolean) take up different amounts of memory, rather than a single “slot.” Further, in python, the name and the data of a variable are actually stored in different places. but thinking of each piece of data as being stored in a single slot in memory provides a good *mental model*—a good way to think about what a variable is.

Let’s look at an example of what a variable is and how it might be used. Imagine you are playing a computer game. A game typically has to keep track of your score. To do this, a programmer writes code that creates a variable, gives the variable a name, and puts some starting value into the variable. In a game, a score typically starts with a value of 0, and the value of the variable changes over time. As the game is played, every time something good happens in the game, the programmer’s code may add to the value

of the variable. If the game calls for it, when something bad happens in the game, the programmer’s code could subtract from the value of the variable.

In Figure [2-3](#), I have arbitrarily chosen slot 3 in memory as the location where the variable score should be saved. Notice that slot 3 is named score and it has a value of 0 in it. In fact, Python makes the choice of where in memory to store data. Because you will always refer to a variable by name, you don’t care where in memory the variable is stored. In this example, whenever you use the name score, Python will use the memory location 3.

21



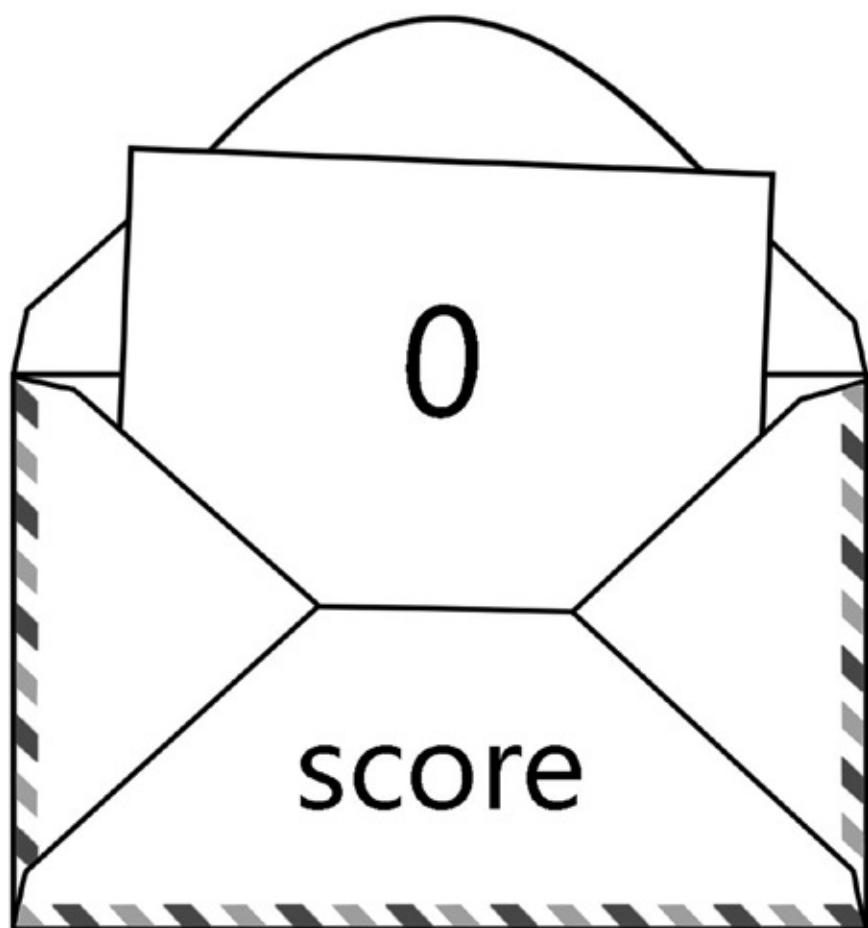
Chapter 2 Variables and assignment statements

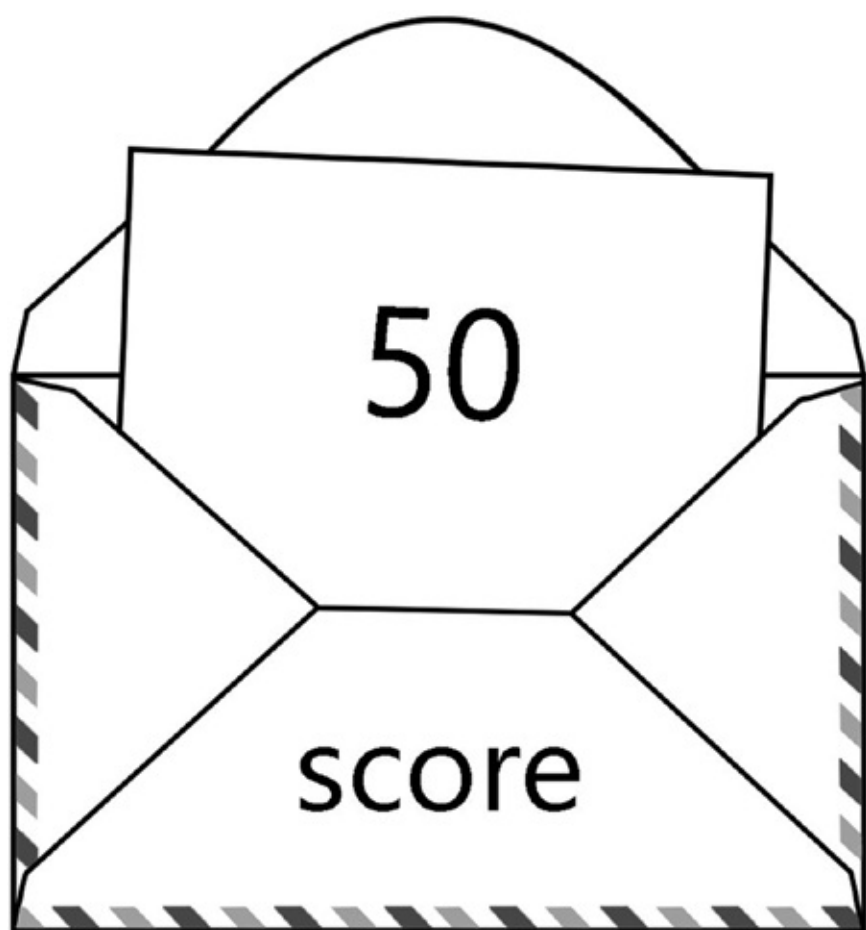
Figure 2-3. Random access memory diagram with one variable defined

Another way to think of a variable is as an envelope or a box into which you can put a value. In this way of thinking, a variable is a *container*—a storage space—with a name.

The contents are the value. The name never changes, but the contents can change over time.

Using the example of a score, imagine that we have an envelope or box with the name *score* on it. Inside, we put the contents—a value. Let's start off with a value of 0, as shown in Figure [2-4](#).





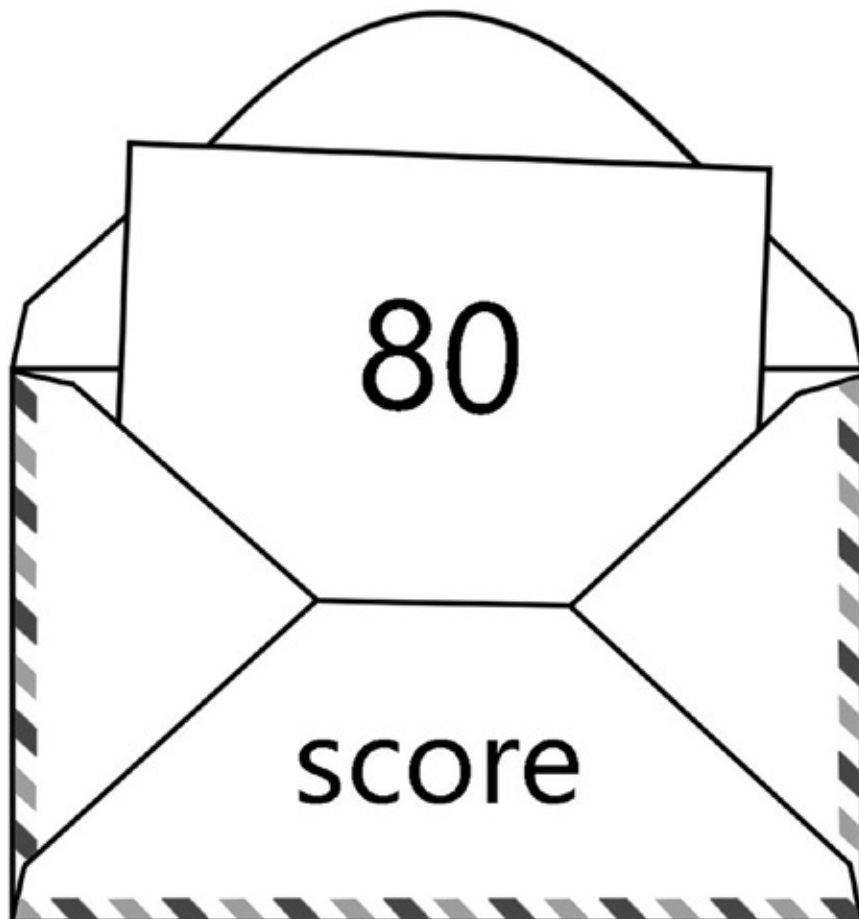
Chapter 2 Variables and assignment statements

Figure 2-4. Visualization of a variable as an envelope

If the user does something good in the game (kills a bad guy, makes a good shot, finds a hidden item, and so on), the user gains 50 points. We take the current value (contents) of the score, which is 0, and add 50. The value of score becomes 50, as shown in Figure [2-5](#).

Figure 2-5. Visualization of a variable with a different value

23



Chapter 2 Variables and assignment statements

Let's say the user does something else good and is awarded another 30 points.

We take the current value of 50 and add 30 to it, giving us a total of 80, as shown in [Figure 2-6](#). So we have the variable called score (which is actually a memory location), and its value is changing over time. The program remembers the current value by having it stored in a variable.

Figure 2-6. Visualization of a variable with yet a different value

Assignment Statements

I have talked about variables and how they are used to store data, but I haven't shown you yet how to use a variable in Python. Let's do that right now.

So much for the theory. In Python, you create and give a value to a variable with an assignment statement.

Definition an

assignment statement is a line of code in which a variable is given a value.

An assignment statement has this general form:

`<variable> = <expression>`

24

Chapter 2 Variables and assignment statements

When I put things in less-than and greater-than brackets, like this `<variable>`, it means that you should replace that whole thing (including the brackets) with something that you choose. Anything written like `<variable>` is a placeholder.

It works like this: the `<expression>`, or everything on the right side of the equals sign, is evaluated, and a value is computed. The resulting value is *assigned* to

(put into) the variable on the left.

This is best explained with some simple examples. Try entering these lines into the Python Shell:

```
>>> age = 29

>>> name = 'Fred'

>>> alive = True

>>> gpa = 3.9

>>>
```

Notice that as soon as you typed an opening quotation mark (such as in typing Fred as a value for the variable name), IDLE recognizes that you are typing a string and turns all characters green until you type the matching closing quote.

Also notice that when you typed the word True, it turned a color (probably purple).

This is an indication that Python has recognized a special word.

When Python runs (or *executes*) assignment statements like these, it first looks to see whether the variable to be assigned was previously used. If the variable has never been seen before, Python allocates an empty slot of memory, attaches a name to it, and puts in the given value. Therefore, when you entered the following line and pressed Return or Enter, Python first looked to see if it had ever seen the variable name age before:

```
>>> age = 29

>>>
```

Since it had not, it allocated a memory slot somewhere (again, we don't care where) attached the age label to it, and then put the value 29 into that memory slot. A similar sequence happened for the other variables.

In pure computer programming terms, the equals sign is not called “equals,” it is

called the *assignment operator*. In an assignment statement, everything on the right of the equals sign is calculated, and the result is *assigned* to the variable on the left.

25

Chapter 2 Variables and assignment statements

Whenever you see an assignment statement, you can read or think of the equals sign as meaning any of the following:

- “is assigned”
- “is given the value of”
- “is set to”
- “becomes”
- “gets”

For example, it might be more helpful and clearer to you to read the line

```
>>> age = 29
```

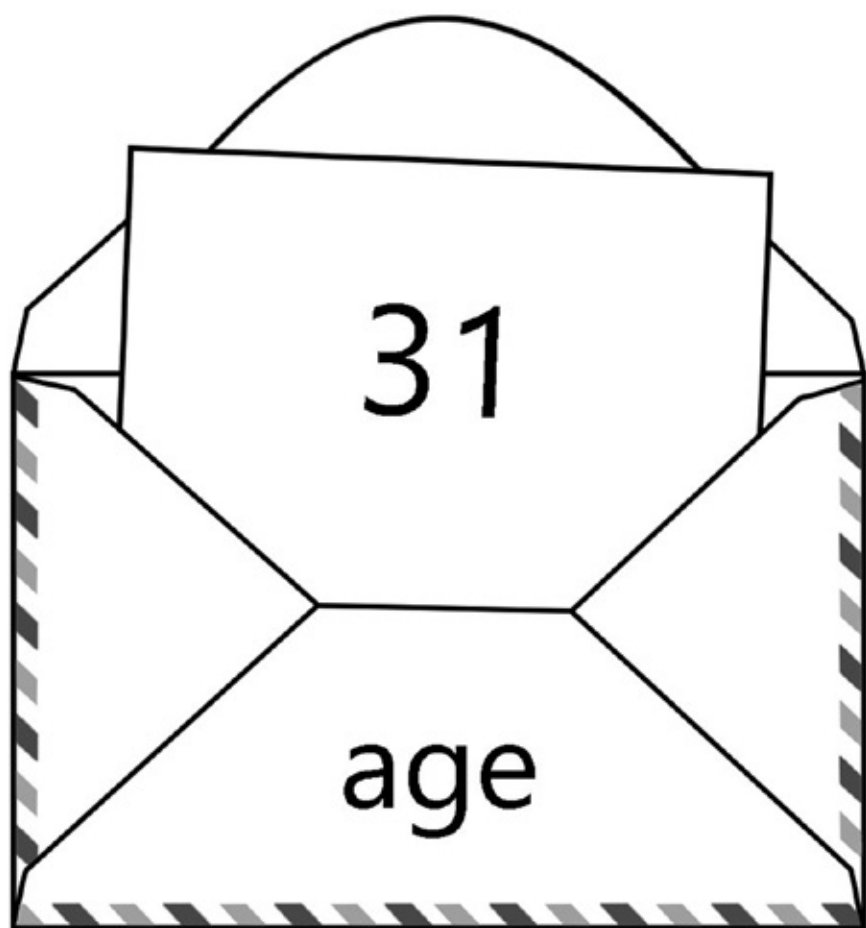
as “age is assigned 29,” or “age is given the value of 29,” or “age is set to 29” or “age becomes 29.”

After executing that line, enter the following line and press Return or Enter:

```
>>> age = 31
```

Python does the same sequence of steps, but now it finds that there already is a variable named `age`. Rather than create a new variable, Python overwrites the current value of the variable `age` with the new value of 31, as shown in [Figure 2-7](#). If you remember the conceptual way of representing a variable as a container (for example, as an envelope), think of this line as replacing the old value inside the envelope with a new value. The variable name `age` stays the same, but the contents change.

26



Chapter 2 Variables and assignment statements

Figure 2-7. *Visualization of a variable as the result of an assignment statement*
Variable Names

By definition, every variable must have a name. It is best to make variable names as descriptive as possible. For example, let's say I was building a virtual aquarium. I would use a variable to keep track of the number of fish in my aquarium. Python doesn't care what you use for a variable name. You could use a name as simple as `x`, or you could use some odd sequence of characters, such as `xddqfmmp`. Or you could create a name like `numberOfFishInAquarium`. That name is much clearer. Names like that make code much more readable and understandable in the long run.

In Python (and all computer languages), there are some rules about naming a variable, though. Here are Python's rules about the name of a variable:

- Must start with a letter (or an underscore)
- Cannot start with a digit
- Can have up to 256 total characters
- Can include letters, digits, underscores, dollar signs, and other characters
- Cannot contain spaces
- Cannot contain math symbols (+, -, /, *, %, parentheses)

27

Chapter 2 Variables and assignment statements

We've seen some examples of legal names, such as `age`, `name`, `score`, `alive`, and `numberOfFishInAquarium`. Here are some examples of illegal names:

- `49ers` (starts with a digit)

- table+chairs (contains a plus sign)
- my age (contains a space)
- (coins) (uses parentheses)

Naming Convention

Definition a

convention is an agreed-upon way of doing things.

In the real world, we have many examples of conventions. When we want to get in an elevator, we let people get out of the elevator before we get in. We always shake hands with our right hand. When we answer a phone, the convention is to say “Hello.”

In the United States, we drive on the right-hand side of the road. That one is not only a convention, it’s the law.

In programming, you can create any variable name you want as long as it follows the rules. However, when creating variable names, I strongly encourage you to use a *naming convention*, by which I mean a consistent approach for creating names for variables. If you create a name like score, where the name is just one word, the convention is to use all lowercase letters.

However, we often want to create descriptive names made by putting together two

or more words. Take, for example, the variable name numberOfFishInAquarium, which is created by putting together five words. In the Python world, there seem to be two common naming conventions.

The first naming convention, and the one that I prefer, is called *camelcase*. The rules of the camelcase convention are very simple:

- The first word is all lowercase.
- For every additional word in the name

- Make the first letter uppercase.
- All other letters are lowercase.

28

Chapter 2 Variables and assignment statements

Here are some examples of variable names that follow the camelcase naming convention:

someVariableName

anotherVariableName

countOfBadGuys

computerScore

humanScore

bonusPointsForCollectingMushrooms

The term *camelcase* describes the way variable names look when using this convention. When an uppercase letter starts a new word, it looks like the hump on a camel. Notice how names that follow the camelcase naming convention are easy to read.

There is another convention that many Python programmers use, which is to separate words with underscores:

this_is_a_variable_name

number_of_fish_in_aquarium

But to me, that's more difficult to both read and write. I am showing this alternative convention here because if you look at other people's code written in Python, you will probably see variable names written with underscores like that.

I have used the camelcase naming convention for years and I will use camelcase

I have used the camelcase naming convention for years and I will use camelcase

throughout this book. If you are coding on your own, obviously you can use whatever names you want, but it really is a good idea to be consistent when naming variables. If you do programming in a class or for a company, the teacher or the company may insist on a particular naming convention so that all programmers can easily understand each other's code.

Keywords

I'm sure that you have heard that computers only understand ones and zeros. This is true. When you write code in Python, Python cannot run or execute your code directly.

Instead, it takes the code that you write and *compiles* it. That is, Python converts your code into a language of ones and zeros that the computer *can* understand. Every language has such a compiler. When the Python compiler reads your code, it looks for special words called *keywords* to understand what your code is trying to say.

29

Chapter 2 Variables and assignment statements

Note python technically has an *interpreter* that turns your code into a machine-independent *byte code*. but to make things simple, i'll refer to this as the python compiler.

The following is a list of the Python keywords. Don't worry about the details right now. I'm just showing you these now to let you know that you cannot use these words as a variable name:

and as assert break class

continue def del elif else

except finally for from global

if import in is lambda

nonlocal not or pass raise

return try while with yield

False None True

You have already seen two of these keywords. The words True and False are the only allowable values for a Boolean variable. True, False, and None are the only keywords that begin with an uppercase letter. There are also some words that Python reserves, like the word print that you saw in the Hello World program. Whenever you type a Python keyword or reserved word, IDLE changes the color of the word. Earlier, when you typed the following, the word print turned orange or purple:

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

This is IDLE looking over your shoulder and telling you that this is one of Python's reserved words.

Caution the important thing to learn here is that you *cannot* use any of these words as a variable name. if you attempt to do so, the python compiler will generate an error message when it tries to compile your program.

30

Chapter 2 Variables and assignment statements

Case Sensitivity

The computer language C was developed in the early 1970s. It was one of the first high-level computer languages and has exerted a great deal of influence on many current computer languages. Languages such as C++, JavaScript, ActionScript (the language of Flash), and Java can all trace their roots to C. There are many similarities among these languages, though each one has a different purpose.

As I mentioned, all computer languages have a compiler that changes the code that you write in that language into lower-level instructions (based on ones and zeros) that the computer really understands. This compilation step happens before you run your program. When C was created, computers were very slow. The people who created C

wanted the C compiler to be as fast as possible. One way they made it fast was to enforce a rule that said that variable names and keywords would be case sensitive—that is, case matters. As humans, we could certainly recognize if our names were spelled with varying degrees of uppercase or lowercase. If I saw my name written as *irv*, *Irv*, or *IRV*, I would know that someone was talking about or to me.

However, because of the need for speed in the C compiler to read and understand

a programmer's variable names, a variable named `myVariable` is *not* the same as one named `myvariable` and is *not* the same as one named `MyVariable`. Each of these represents a unique variable.

Python has this same trait. Variable names and keywords in Python are all case sensitive. For example, `print` is not the same as `Print`. This will bite you many times over. You will spend a great deal of time scratching your head about why your program won't compile, only to realize hours later that you used a lowercase letter where you needed an uppercase one.

Tip this is another great reason to following a strict naming convention. if you follow a naming convention such as camelcase, you will make fewer uppercase/lowercase naming errors.

31

Chapter 2 Variables and assignment statements

More Complicated Assignment Statements

Now that we have an understanding of the rules of naming variables and a naming

convention that will help us name variables, let's look at some more details of assignment statements.

Remember, this is the general form of an assignment statement:

<variable> = <expression>

So far, I've only shown assignment statements that give a variable a simple value. But the <expression> part on the right side of the equals sign (actually called the *assignment operator*) can be as simple or as complicated as you need it to be. The right-hand side can also contain variables. Here's an example:

```
>>> myAge = 31
```

```
>>> yourAge = myAge
```

```
>>>
```

The first line creates and sets a variable named `myAge` to 31. The second line creates a variable named `yourAge` and sets it to the current value of the variable `myAge`. After running these two lines, both variables would be set to the value 31.

An assignment statement computes the value of whatever is on the right-hand side of the equals sign and assigns it to the variable on the left-hand side. Whenever a variable appears in an expression (that is, on the right-hand side of an assignment statement), the expression is evaluated (computed) by using the value of each variable. In other words, as the programmer, you write the name of a variable, but when the statement runs, the computer uses the current value of that variable at that time.

Here's a simple example of doing some addition:

```
>>> numberOfMales = 5
```

```
>>> numberOfFemales = 6
```

```
>>> numberOfPeople = numberOfMales + numberOfFemales
```

```
>>>
```

First, we use two assignment statements to create two variables: `numberOfMales` and `numberOfFemales`. In the third line, we see those two variables on the right side of the equals sign (the assignment operator). To generate a result, Python uses the value of those variables, 5 and 6, does the math, gets an answer of 11, and assigns that result into the variable on the left side of the equals sign.

and assigns that result into the variable on the left side of the equals sign: numberOfPeople.

32

Chapter 2 Variables and assignment statements

I want to make it very clear that the equals sign in an assignment statement is very different from the way an equals sign is used in math. In math, the equals sign implies that the things on the left side and the right side have the same value. In Python, that's not the case.

To drive this point home, consider these two lines of code:

```
>>> myAge = 25

>>> myAge = myAge + 1

>>>
```

If you are a mathematician, the second line will jump out at you as an impossibility—

that is, there is no value of myAge for which that statement is true.

However, this is *not* an equation; it is an assignment statement. Here is what the second line says (starting on the right-hand side):

1. Take the current value of the variable myAge
2. Add 1 to it
3. Put the resulting value back in to the variable myAge

This statement effectively changes the value of the variable myAge by adding 1 to it.

Using a variable to count something is done all the time in programming, and it is very common to see lines like this in code.

Print Statements

Now, how do we know if things are working? We would like to have a way to reach into a variable and see the value inside. Remember the print statement from our Hello World program? The print statement is very general purpose. You ask it to print something and it prints whatever you ask it to print into the Shell window. The general print statement looks like this:

```
print(<whatever you want to see>)
```

33

Chapter 2 Variables and assignment statements

Here are some examples in the Shell:

```
>>> eggsInOmlette = 3
```

```
>>> print(eggsInOmlette)
```

3

```
>>> knowsHowToCook = True
```

```
>>> print(knowsHowToCook)
```

True

```
>>>
```

The print statement can also print multiple things on a single line. You can do this by separating the items you want to print with commas. When the print statement runs, each comma is replaced by a space:

```
>>> print(eggsInOmlette, knowsHowToCook)
```

3 True

```
>>>
```

You can use this to nicely format your output. For example, it allows you to print a description of what variable you are printing:

```
>>> eggsInOmlette = 3
```

```
>>> eggsInOmlette = 3
>>> print('eggsInOmlette is:', eggsInOmlette)
eggsInOmlette is: 3
>>> knowsHowToCook = True
>>> print('knowsHowToCook is:', knowsHowToCook)
knowsHowToCook is: True
>>> print('eggsInOmlette and knowsHowToCook are', eggsInOmlette, 'and',
knowsHowToCook)
eggsInOmlette and knowsHowToCook are 3 and True
>>>
```

Here are more examples of assignment statements and print statements, using all four types of data:

```
>>> numberInADozen = 12
>>> print('There are', numberInADozen, 'items in a dozen')
There are 12 items in a dozen
>>> learningPython = True
```

34

Chapter 2 Variables and assignment statements

```
>>> print('It is', learningPython, 'that I am learning Python')
It is True that I am learning Python
>>> priceOfCandy = 1.99
>>> print('My candy costs', priceOfCandy)
```

My candy costs 1.99

```
>>> myFullName = 'Irv Kalb'
```

```
>>> print('My full name is', myFullName)
```

My full name is Irv Kalb

```
>>>
```

There is one additional note about the print statement. To make things a little

clearer in your output, you may want to include one or more blank lines. To create a blank line of output, you can use an empty print statement. Just write the word print with a set of open and close parentheses, like this:

```
>>> print()
```

Note in python 2, the print statement has a different form (syntax). in python 2, the print statement did not require the parentheses around the item(s) that

you want to print. it looked like this: print <item1>, <item2>, ... this is perhaps the most noticeable difference between python 2 and python 3. if

you see code elsewhere written in python 2 that is missing parentheses in print

statements, you can often modify these statements to work in python 3 by adding

an outermost set of parentheses.

Simple Math

Now let's move on to some simple math for use in assignment statements.

Python

recognizes the following set of math operators:

- + Add
- - Subtract
- * Multiply

- * Multiply

- / Divide

35

Chapter 2 Variables and assignment statements

- // Integer Divide

- ** Raise to the power of

- % Modulo (also known as *remainder*)

- () Grouping (we'll come back to this)

Let's try some very simple math. For demonstration purposes, I'll just use variables named x and y. In the Shell, try the following:

```
>>> x = 9
```

```
>>> y = 6
```

```
>>> print(x + y)
```

15

```
>>> print(x - y)
```

3

```
>>> print(x * y)
```

54

```
>>> print(x / y)
```

1.5

```
>>> print(x // y)
```

1

Note in python 2, the divide operator worked differently. if you divided an integer by an integer, you got an integer as a result. For example if you divided 9 by 6, you got 1. the behavior of the divide operator was changed in python 3 to always give a floating-point answer, and the integer divide operator (with two slashes) was added.

COMPUTERS CAN REPRESENT INTEGERS PERFECTLY

as humans, we represent integers using base 10 (digits from 0 to 9). Computers represent integers using base 2 (using only 1 and 0). but there is an exact mapping between the two bases. For every base 10 number, there is an exactly equivalent base 2 number. however, because of the way computers represent floating-point numbers, this is not the case for floating-point numbers. there is no such mapping between base 10 fractions and base 36

Chapter 2 Variables and assignment statements

2 fractions. When representing floating-point fractional numbers, there is often some small amount of rounding; that is, floating-point fractional numbers are a close approximation of the intended number. For example, if we attempt to divide 5.0 by 9.0, we see this:

```
>>> print(5.0 / 9.0)
```

```
0.555555555556
```

the decimal values goes on forever, but when represented as a float, the value gets rounded off.

Let's try out the last two math operators. "Raise to the power of" is very straightforward. In the following code, we want to raise x to the power of y:

```
>>> x = 2
```

```
>>> y = 3
```

```
>>> print(x ** y)
```

```
>>>
```

Finally, there is modulo. The *modulo operator*—the percent sign—gives you the remainder of a division. With an integer division, the result is just the integer result. The modulo operator allows you to get the remainder. Here's an example. Imagine that you have a puppy that is 29 months old. Using an integer divide and the modulo operator, we can do an easy conversion to find out the age of the puppy is in years and months.

```
>>> ageInMonths = 29
```

```
>>> years = ageInMonths // 12
```

```
>>> months = ageInMonths % 12
```

```
>>> print("My puppy's age is", years, "years and", months, "months.") My  
puppy's age is 2 years and 5 months.
```

```
>>>
```

If we reverse the process, you can see how we can get back to the original number:

```
>>> puppysAge = (years * 12) + months
```

```
>>> print("Puppy's age in months is:", puppysAge)
```

```
Puppy's age in months is: 29
```

```
37
```

Chapter 2 Variables and assignment statements

Order of Operations

Back in elementary school, in a lesson about math, my teacher went through a long description of a topic called the *order of operations*. We were told that some math operators had precedence over other ones. For example, look at this assignment

statement:

$$x = 5 + 4 * 9 / 6$$

What operations are done in what order? The teacher explained that the acronym PEMDAS would help us to figure out the order. PEMDAS described the precedence order as follows:

1. Parentheses
2. Exponents
3. Multiplication
4. Division
5. Addition
6. Subtraction

However, I thought that it was a terrible idea to have some implied, seemingly arbitrary ordering of math operators. Let's look at the assignment statement again: $x = 5 + 4 * 9 / 6$

You must understand the PEMDAS ordering to figure it out. First, you would multiply 4 by 9, take the result and divide that by 6, and then add 5, before storing the answer in x.

Because of my conviction for clarity, I feel that writing an assignment statement like this is an extremely poor technique. You are writing in a way that forces future readers to have an understanding of PEMDAS in order to infer what you meant by this statement.

Instead, it would be much clearer to both you and future readers if you were to use parentheses to group operations. Using parentheses allows you to "force" the order of operations so that the steps happen in whatever order you want. If you wanted to write the line in a way that reflects what would happen following

PEMDAS, you would write:

PEMDAS, it would look like this:

$$x = 5 + ((4 * 9) / 6)$$

38

Chapter 2 Variables and assignment statements

When you have nested sets of parentheses, the only rule you need to know is that sets of parentheses are evaluated from the innermost set to the outmost set. In this example, $(4 * 9)$ is evaluated first, that result is then divided by 6, and then 5 is added to that result.

If you wanted the operations performed in a different order, you could use parentheses to create different groupings. For example:

$$x = (5 + 4) * (9 / 6)$$

These parentheses say that you should add 5 and 4, divide 9 by 6, and then multiply the results.

Tip adding parentheses as in the preceding statements makes your intent much clearer and does not rely on the reader to understand the pemdas rules. I strongly encourage you to add parentheses like these to force the order of operations.

First Python Programs

Let's take everything we've learned in this chapter and write some very small but useful Python programs. We'll start by writing a simple program to add up the value of all the one-dollar bills and five-dollar bills that are in a wallet. Start by opening a new Python editor window (Control+N (Windows) or Command+N (Mac), or File ► New). This is

what that code could look like:

```
numberOfOneDollarBills = 3
```

```
numberOfFiveDollarBills = 2
```

```
total = numberOfOneDollarBills + (numberOfFiveDollarBills * 5)
```

```
print('Total amount is: ' + total)
```



```
print('Total amount is: ', total)
```

Again, none of these lines execute immediately; they have all just been entered into a file. In order to see any results, we have to run (execute) the program we have just written.

First, save the file (press Control+S (Windows) or Command+S (Mac) or click File ►

Save). The first time you save a new file like this, you must give it a name. All Python file names should end in .py, so name this file something like MoneyInWallet.py.

39

Chapter 2 Variables and assignment statements

Now that the file is saved, you are ready to run the program by pressing the F5 shortcut key or clicking Run ► Run Module. If there are no errors in your program, you will see the output of your program show up in the Shell. You should see this:

Total amount is: 13

If you had any errors, read the bottom line of the error message, identify what you typed incorrectly, fix it, save, and run again.

Let's build another simple program. In IDLE, open a new file (Command/

Control+N). This time we'll write a program to calculate how much money you should be paid for working at a job. For the first 40 hours, you should be paid at an hourly rate.

Just like any other type of data, a list is created using an assignment statement. That is, you write single variable name followed by an equals sign, and then you define your list.

```
<myListVariable> = [<element>, <element>, ... <element>]
```

A list can essentially have any number of elements. The actual number of elements is limited only by the amount of memory available. Here are some examples: `shoppingList = ['apples', 'bananas', 'cherries', 'dates', 'eggplant']`

```
scoresList = [24, 33, 22, 45, 56, 33, 45]
```

A list can also be created using a mix of data types:

```
mixedList = [True, 5, 'some string', 123.45]
```

Note that we are showing variable names that represent a list in the form of

`<name>List`. This is not required, but a name in this form clearly indicates that the variable represents a list rather than an individual piece of data. We will use this naming convention throughout the rest of this book.

186

Chapter 7 Lists

A list is a new data type. To show that a list is a standard data type in Python, let's create a list, print it, and use the `type` function to find out which data type it is: `>>> mixedList = [True, 5, 'some string', 123.45]`

```
>>> print(mixedList)
```

```
[True, 5, 'some string', 123.45]
```

```
>>> print(type(mixedList))
```

```
<class 'list'>
```

```
>>>
```

So, a list is of type list, independent of the type of data of its contents. A list is unusual because it is made up of multiple pieces of data, where each one can be of any data type.

Empty List

Although a list can have any number of elements, there is also a special list that is made up of no elements. This is known as the *empty list*: `>>> someList = []` # set a list variable to the empty list – no elements

```
>>> print(someList)
```

```
[]
```

```
>>>
```

You can think of the empty list relative to other lists, like zero in comparison to other numbers. We will use this later by creating an empty list and then adding elements to it on the fly.

Position of an Element in a List: Index

You've seen that we can create a list with the square bracket syntax, and we can print a list using the print statement, but the power of a list comes from the ability to use the individual elements in the list. Therefore, we need a way to *reference* (get at) any individual element of a list. Let's look at our example shopping list again: `>>> shoppingList = ['apples', 'bananas', 'cherries', 'dates', 'eggplant']`

```
>>> print(shoppingList)
```

```
['apples', 'bananas', 'cherries', 'dates', 'eggplant']
```

```
>>>
```

187

Chapter 7 Lists

You can think of any physical list, like a shopping list, as a numbered list. That is, we could assign a consecutive integer to each element, and reference any

element in our shopping list using that number. In fact, that identifying number has a clear definition.

Definition an

index is the position (or number) of an element in a list. (it is sometimes referred to as a *subscript*.)

An index is always an integer value. Because each element has an index (number), we can reference any element in the list by using its associated number or position in the list.

To us humans, in our shopping list, apples is element number 1, bananas is element number 2, and cherries is element number 3. This is the way that we typically think of numbering things: Sample shoppingList:

Human

Number Element

1 'apples'

2 'bananas'

3 'cherries'

4 'dates'

5 'eggplant'

In Python (and most other computer languages), the elements in a list are also

numbered consecutively—but the counting starts at zero. That is, all lists start at an index of zero. The indices for the preceding list are 0, 1, 2, 3, and 4. 'apples' is element number 0, 'bananas' is element number 1, and up to 'eggplant', which is element 4: Python Index Element

0 'apples'

1 'bananas'

2 'cherries'

3 'dates'

4 'eggplant'

188

Chapter 7 Lists

This list has five elements, but they are numbered 0 to 4.

Caution this concept of starting a count at zero is very important, and until you wrap your head around it, it will cause you much grief!

Accessing an Element in a List

Now we have a way to represent a list of data in a single variable: enclosing the list in brackets, separating elements by commas. But we need some way to *get at* the individual elements in the list. The way we do that is to use the following syntax: <listVariable>[<index>]

This syntax results in the value of the element in the list at the given index. Let's assume a list of numbers defined with this assignment statement: numbersList = [20, -34, 486, 3129]

We can access each element in the numbersList as follows:

numbersList[0] # would evaluate to 20

numbersList[1] # would evaluate to -34

numbersList[2] # would evaluate to 486

numbersList[3] # would evaluate to 3129

I'll demonstrate this in the Shell using our shopping list with some simple print statements:

```
>>> shoppingList = ['apples', 'bananas', 'cherries', 'dates', 'eggplant']
```

```
... ..
```

```
>>> print(shoppingList)
['apples', 'bananas', 'cherries', 'dates', 'eggplant']
>>> print(shoppingList[2])
cherries
>>> print(shoppingList[4])
eggplant
>>> print(shoppingList[0])
apples
>>>
```

189

Chapter 7 Lists

Here are some suggestions for how to read a list variable with an index value. Let's say you see something like this: `myList[2]`

Rather than read it as “`myList` bracket 2 bracket,” it is probably clearer if you read it as any of the following:

- `myList` element 2

- The element in position 2 of `myList`
- Element 2 of `myList`
- The third element of `myList`
- `myList` sub 2 (“old school” reference to subscripts)

The first one—“`myList` element 2”—is probably the most straightforward.

Using a Variable or Expression as an Index in a List

An index can also be written as a variable or an expression. In fact, most of the time, we access elements in a list this way. The following is a simple code

snippet that demonstrates this approach. We'll ask the user to enter an integer and will use that value as an index to our shopping list: >>> shoppingList = ['apples', 'bananas', 'cherries', 'dates', 'eggplant']

```
>>> myIndex = input('Enter an index: ')
```

Enter an index: 3

```
>>> myIndex = int(myIndex) # convert to integer
```

```
>>> myElement = shoppingList[myIndex] # use as an index into the list
```

```
>>> print('The element at index', myIndex, 'is', myElement)
```

The element at index 3 is dates

Let's show this as a simple program with a loop. We'll use concepts from [Chapter 6 to](#)

allow the user to run the program multiple times:

```
shoppingList = ['apples', 'bananas', 'cherries', 'dates', 'eggplant']
```

```
while True:
```

```
    myIndex = input('Enter a number to use as an index: ')
```

```
    190
```

Chapter 7 Lists

```
    if myIndex == ":
```

```
        break
```

```
    myIndex = int(myIndex)
```

```
    myElement = shoppingList[myIndex]
```

```
    print('The element at index, myIndex, 'is', myElement)
```

```
    print('Bye')
```

```
print myC )
```

Entering any value between 0 and 4 gives us the appropriate answer:

Enter a number to use as an index: 0

The element at index 0 is apples

Enter a number to use as an index: 1

The element at index 1 is bananas

Enter a number to use as an index: 2

The element at index 2 is cherries

Enter a number to use as an index: 3

The element at index 3 is dates

Enter a number to use as an index: 4

The element at index 4 is eggplant

Now run it again, but this time let's see what happens if we enter 100 as the index: Enter a number to use as an index: 100

Traceback (most recent call last):

File "*Learn to Program with Python*Chapter 7 Lists/Kalb Chapter 7 Code/

IndexAsVariable.py", line 10, in <module>

```
myElement = shoppingList[myIndex]
```

IndexError: list index out of range

Again, when you get a runtime error or a traceback, you should read the last line first.

It tells you which type of error occurred. This error message says, "Index Error: list index out of range." It is extremely clear. You tried to access an element that

is outside the valid range of the list indices. There is no element 100, so when you try to use that as an index, you get an error. Python has built-in range checking to ensure that you are using a valid number when you attempt to index into a list.

191

Chapter 7 Lists

Note Many other languages do not do range checking. if you use an out-of-range index in one of those languages, the code accesses a part of the memory of the computer that is not part of the list, and retrieves some arbitrary value found there. sometime later, when you attempt to use the value, the program may crash mysteriously. tracking down errors like that can be extremely difficult.

Changing a Value in a List

So far, I've shown list index references on the right-hand side of an assignment statement. This is how you *get* (or *retrieve*) a value from a list. You can also *set* a value in a list—that is, replace the current contents of an element in a list by putting the list variable with its index on the left-hand side of an assignment, like this: `>>> shoppingList = ['apples', 'bananas', 'cherries', 'dates', 'eggplant']`

```
>>> shoppingList[2] = 'cucumbers'
```

```
>>> print(shoppingList)
```

```
['apples', 'bananas', 'cucumbers', 'dates', 'eggplant']
```

```
>>>
```

This changes the value of an element at the given index to a new value. Notice that element 2 was 'cherries' but has been changed to 'cucumbers'.

Now you know how to change the value of a given element. Shortly, I'll show you

how to change the number of elements in a list. Python people talk about lists as being *mutable*, which means changeable.

Using Negative Indices

In addition to indices starting at 0 and going up to the number of elements minus 1, there is another way to index elements in a list. Python allows you to use negative integers as indices to a list. A negative index means to count backwards from the end, the end being the number of elements in the list. Here are the positive and equivalent negative indices for a list of five elements: 192

Chapter 7 Lists

0 -5 <element>

1 -4 <element>

2 -3 <element>

3 -2 <element>

4 -1 <element>

Let's demonstrate with our shopping list:

```
>>> shoppingList = ['apples', 'bananas', 'cherries', 'dates', 'eggplant']
```

```
>>> print(shoppingList[-1])
```

eggplant

```
>>> print(shoppingList[-2])
```

dates

```
>>> print(shoppingList[-3])
```

cherries

When using a negative number as an index, Python takes the number of elements in the list and then adds the negative amount to get the positive index. Using our shopping list as an example, element -2 is 5 (the number of elements in the list) minus 2, which equals 3, for a value of 'dates'. Negative indexing is rarely used, but the main way to use it is to use -1 as an index as a quick way of getting to the last element in a list.

Building a Simple Mad Libs Game

Let's build an old, popular game: Mad Libs. We'll start by getting input from the user, just like in a real Mad Libs game, and use the user's responses in our story. The starting version of this game has nothing to do with lists, but once we build the base game, we'll modify it to use lists.

The starting version of this program is all about strings. Remember that when we want to add strings together, it is called *concatenation*. And the concatenation operator is the plus sign between strings. Just as we can add a long group of numbers, we can also concatenate multiple strings. In this version of Mad Libs, our story is just one sentence that is built using concatenation. Our one-sentence story will be as follows: <name> <verb> through the forest, hoping to escape the <adjective> <noun>.

```
# MadLib (version 1)
```

```
while True:
```

```
    name = input('Enter a name: ')
```

```
193
```

```
Chapter 7 Lists
```

```
    verb = input('Enter a verb: ')
```

```
    adjective = input('Enter an adjective: ')
```

```
    noun = input('Enter a noun: ')
```

```
    sentence = name + ' ' + verb + ' through the forest, hoping to escape
```

```
    the ' + adjective + ' ' + noun + '.'
```

```
    print()
```

```
    print(sentence)
```

```
    print()
```

```
    ... - ... - ...
```

```
# See if the user wants to quit or continue
```

```
answer = input('Type "q" to quit, or anything else (even Return/Enter) to  
continue: ')
```

```
if answer == 'q':
```

```
    break
```

```
print('Bye')
```

The program asks the user to enter the four parts of speech and then concatenates the sentence and prints it. Here's what our program looks like when it runs: >>>

Enter a name: Weird Al Yankovic

Enter a verb: screams

Enter an adjective: orange

Enter a noun: dinosaur

Weird Al Yankovic screams through the forest, hoping to escape the orange
dinosaur.

Return/Enter to continue, "q" to quit:

Enter a name: The Teenage Mutant Ninja Turtles

Enter a verb: burped

Enter an adjective: frilly

Enter a noun: Frisbee

194

Chapter 7 Lists

The Teenage Mutant Ninja Turtles burped through the forest, hoping to

escape the frilly Frisbee.

Return/Enter to continue, "q" to quit: q

Bye

>>>

Adding a List to Our Mad Libs Game

Now, we'll change the program. Rather than have the user enter a name, we'll build a pool of names and select one randomly for the user. The pool of predetermined names will be built as a list. We could use any names for our list, but to make our Mad Libs game fun, our list will look like this: `namesList = ['Weird Al Yankovic', 'The Teenage Mutant Ninja Turtles',`

`'Supergirl', \`

`'The Stay Puft Marshmallow Man', 'Shrek', 'Sherlock Holmes', \`

`'The Beatles', 'Powerpuff Girl', 'The Pillsbury Doughboy']`

Next, we want to choose a random name from this list. This particular list has nine elements in it. In order to select a random element from the list, we need to generate a random index between 0 and 8 (remember, list indices start at zero). In [Chapter 6](#), we learned that to generate a random number, we use the `randrange` function in the `random` package: `import random` `randomIndex = random.randrange(<lowerLimit>, <upToButNotIncluding>)`

Again, our goal is to select a random number to use as an index of an element in the list. With our list of nine names, we would call `random.randrange`, passing in a 0 and a 9. It would return a random integer of 0 to 8 (up to but not including 9). The resulting program would look like this: `# MadLib (version 2)`

```
import random
```

```
namesList = [ 'Weird Al Yankovic', 'The Teenage Mutant Ninja Turtles',
```

```
'Supergirl', \
```

```
'The Stay Puft Marshmallow Man', 'Shrek', 'Sherlock Holmes', \
```

'The Beatles', 'Powerpuff Girl', 'The Pillsbury Doughboy']

195

Chapter 7 Lists

while True:

nameIndex = random.randrange(0, 9) # Choose a random index into the

namesList

name = namesList[nameIndex] # Use the index to choose a random name

verb = input('Enter a verb: ')

adjective = input('Enter an adjective: ')

noun = input('Enter a noun: ')

sentence = name + ' ' + verb + ' through the forest, hoping to escape

the ' + adjective + ' ' + noun + '.'

print()

print(sentence)

print()

See if the user wants to quit or continue

answer = input('Type "q" to quit, or anything else (even Return/Enter) to
continue: ')

if answer == 'q':

break

print('Bye')

In this version, we added the list of names and we replaced the code that asked the user for a name with code that randomly picks a name from the list provided.

Determining the Number of Elements in a List:

The len Function

The list of names could contain any number of names. Rather than hard-code an integer for the number of names in our list, we would ideally want to write code that would be able to work for any number of elements in the list. Therefore, we need a way to find out how many elements are in a list. Python has a built-in function for this, called `len` (short for *length*).

```
len(<listVariable>)
```

196

Chapter 7 Lists

To find the length of a list—that is, the number of elements in a list—you call the `len` function and pass in the variable name that holds the list: >>>

```
shoppingList = ['apples', 'bananas', 'cherries', 'dates', 'eggplant']
```

```
>>> nElements = len(shoppingList)
```

```
>>> print('There are', nElements, 'items in our shopping list.')
```

There are 5 items in our shopping list.

```
>>>
```

There are five elements in our shopping list, but again, the elements are numbered 0

to 4. If we want to use `random.randrange` to choose a random element, we certainly want to use 0 as the low-end value because indices always start at 0. But `random.randrange` also requires an `<upToButNotIncludingHighEnd>` value. The `len` of a list is perfect for use as the high end with this call because it gives you one more than the last index to the list.

Let's incorporate the `len` function into our Mad Libs program:

```

# MadLib (version 3)

import random

namesList = [ 'Weird Al Yankovic', 'The Teenage Mutant Ninja Turtles',
'Supergirl', \
'The Stay Puft Marshmallow Man', 'Shrek', 'Sherlock Holmes', \
'The Beatles', 'Powerpuff Girl', 'The Pillsbury Doughboy',
'Sam-I-Am']

nNames = len(namesList) # find out how many names are in the list of names

while True:

    nameIndex = random.randrange(0, nNames) # Choose a random index into
    the namesList

    name = namesList[nameIndex] # Use the index to choose a random name

    verb = input('Enter a verb: ')

    adjective = input('Enter an adjective: ')

    noun = input('Enter a noun: ')

    sentence = name + ' ' + verb + ' through the forest, hoping to escape
    the ' + adjective + ' ' + noun + '.'

    print()

197

Chapter 7 Lists

print(sentence)

```



```
print()
```

```
# See if the user wants to quit or continue
```

```
answer = input("Type 'q' to quit, or anything else (even Return/Enter) to  
continue: ' ")
```

```
if answer == 'q':
```

```
break
```

```
print('Bye')
```

Notice that in this version, we've added another name to our list of names. But we also used the len function to set a variable, nNames, to the number of elements in our list of names. Finally, we used that variable in our call to randrange. Using this approach, we can put as many names in our list as we want, and the code will adjust at runtime for us.

Programming Challenge 1

Similar to the modification to use a list of names, let's modify the program to include a list of verbs, a list of adjectives, and a list of nouns. The program should randomly choose a name, a verb, an adjective, and a noun. You can put as many elements as you want into each list, and the program should create and print a fully randomized Mad Lib.

Here is our Mad Libs program using lists for names, verbs, adjectives, and nouns.

I have tried to choose silly words to generate humorous sentences:

```
# MadLib (version 4)
```

```
import random
```

```
namesList = [ 'Weird Al Yankovic', 'The Teenage Mutant Ninja Turtles',  
'Supergirl', \
```

```
'The Stay Puft Marshmallow Man', 'Shrek', 'Sherlock Holmes', \
'The Beatles', 'Powerpuff Girl', 'The Pillsbury Doughboy',
'Sam-I-Am']
```

```
nNames = len(namesList) # find out how many names are in the list of names
```

```
verbsList = ['screamed', 'burped', 'ran', 'galumphed', 'rolled', 'ate',
'laughed', 'complained', 'whistled']
```

198

Chapter 7 Lists

```
nVerbs = len(verbsList)
```

```
adjectivesList = ['purple', 'giant', 'lazy', 'curly-haired', 'wireless
electric', 'ten foot tall']
```

```
nAdjectives = len(adjectivesList)
```

```
nounsList = ['ogre', 'dinosaur', 'Frisbee', 'robot', 'staple gun', 'hot dog
vendor', 'tortoise', 'rodeo clown', 'unicorn', 'Santa hat', 'garbage can']
```

```
nNouns = len(nounsList)
```

```
while True:
```

```
nameIndex = random.randrange(0, nNames) # Choose a random index into
the namesList
```

```
name = namesList[nameIndex] # Use the index to choose a random name
```

```
verbIndex = random.randrange(0, nVerbs)
```

```
verb = verbsList[verbIndex]
```

```
adjectiveIndex = random.randrange(0, nAdjectives)
adjective = adjectivesList[adjectiveIndex]
nounIndex = random.randrange(0, nNouns)
noun = nounsList[nounIndex]
sentence = name + ' ' + verb + ' through the forest, hoping to escape
the ' + adjective + ' ' + noun + '.'
print()
print(sentence)
print()
# See if the user wants to quit or continue
answer = input('Type "q" to quit, or anything else (even Return/Enter) to
continue: ')
if answer == 'q':
    break
print('Bye')
199
```

Chapter 7 Lists

This code generated the following sample output—without any suggestions from the user:

The Pillsbury Doughboy burped through the forest, hoping to escape the giant Frisbee.

Type "q" to quit, or anything else (even Return/Enter) to continue:

Sam-I-Am complained through the forest, hoping to escape the wireless electric ogre.

Type "q" to quit, or anything else (even Return/Enter) to continue:

The Beatles ate through the forest, hoping to escape the lazy staple gun.

Type "q" to quit, or anything else (even Return/Enter) to continue:

The Beatles laughed through the forest, hoping to escape the ten foot tall unicorn.

Type "q" to quit, or anything else (even Return/Enter) to continue:

The Stay Puft Marshmallow Man galumphed through the forest, hoping to escape the giant unicorn.

Type "q" to quit, or anything else (even Return/Enter) to continue: q

Bye

Using a List Argument with a Function

In the prior code, you may have noticed a pattern. For each of the four lists

(nounsList, verbsList, adjectivesList, and nounsList), we have built essentially the same code. Whenever we wanted to select a random element, we chose a random index and then found the element at that index. Although this clearly works, whenever we see essentially the same code repeated, it is a signal that it is probably 200

Chapter 7 Lists

a good candidate to turn into a function. In this case, rather than repeat the same set of operations four times, we'll build a single function to select a random element from a list and call it four times: # MadLib (version 5)

```
import random
```

```
def chooseRandomFromList(aList):  
    nItems = len(aList)  
  
    randomIndex = random.randrange(0, nItems)  
  
    randomElement = aList[randomIndex]  
  
    return randomElement  
  
namesList = [ 'Weird Al Yankovic', 'The Teenage Mutant Ninja Turtles',  
    'Supergirl', \  
    'The Stay Puft Marshmallow Man', 'Shrek', 'Sherlock Holmes', \  
    'The Beatles', 'Powerpuff Girl', 'The Pillsbury Doughboy',  
    'Sam-I-Am']  
  
verbsList = ['screamed', 'burped', 'ran', 'galumphed', 'rolled', 'ate',  
    'laughed', 'complained', 'whistled']  
  
adjectivesList = ['purple', 'giant', 'lazy', 'curly-haired', 'wireless  
electric', 'ten foot tall']  
  
nounsList = ['ogre', 'dinosaur', 'Frisbee', 'robot', 'staple gun', 'hot dog  
vendor', 'tortoise', 'rodeo clown', 'unicorn', 'Santa hat', 'garbage can']  
  
while True:  
  
    name = chooseRandomFromList(namesList)  
  
    verb = chooseRandomFromList(verbsList)  
  
    adjective = chooseRandomFromList(adjectivesList)  
  
    noun = chooseRandomFromList(nounsList)
```

```
sentence = name + ' ' + verb + ' through the forest, hoping to escape
```

```
the ' + adjective + ' ' + noun + '.'
```

```
print()
```

```
print(sentence)
```

```
print()
```

```
201
```

Chapter 7 Lists

```
# See if the user wants to quit or continue
```

```
answer = input("Type \"q\" to quit, or anything else (even Return/Enter) to  
continue: ')
```

```
if answer == 'q':
```

```
break
```

```
print('Bye')
```

In this version, we've built a small function called `chooseRandomFromList`. It is designed to expect to have one parameter passed in when it is called. It is expected to be passed in a list. The `aList` parameter variable takes on the value of the list passed in. We used a very generic name here because we do not know what the contents of the list are, and inside the function, we do not care. The function uses the `len` function to see how many items are in the list, chooses a random index, finds the element at that index, and returns that element. From the main code, we now call the function four times, passing in four different lists. This version of the code generates the same type of Mad Libs sentences as the earlier version, but it is easier to read and is less prone to errors.

It turns out that Python actually provides this exact functionality with a built-in function in the `random` package. The function is called `choice`. To get a randomized selection from a list, you make a call like this:

```
random.choice(<list>)
```

Here is a simple example:

```
>>> optionsList = ['rock', 'paper', 'scissors', 'lizard', 'Spock']
```

```
>>> anOption = random.choice(optionsList)
```

```
>>> print(anOption)
```

```
paper
```

```
>>>
```

202

Chapter 7 Lists

Accessing All Elements of a List: Iteration

Using bracketing syntax such as `someList[someIndex]`, we have a way to access any element in a list. But we need a way to access *all* elements in a list. As a simple example, let's say we just wanted to print out the value of all the elements of a list. We can write this: `print(myList)` And that works fine. But it prints the list in the Python list syntax (including the square brackets and commas), and prints all elements horizontally. What if we wanted to print one element per line? Or what if the list contains numbers and we want to add them up? We need some way to get at all the elements of a list, but one at a time. That's called iteration.

Definition *Iterate* means to traverse through, or visit all elements of a list.

Using code that we already know, we can build a loop using a `while` statement to get the job done. The following is some code to print our shopping list, one item per line.

(This is for demonstration purposes only—it is *not* the best way.)

```
shoppingList = ['apples', 'bananas', 'cherries', 'dates', 'eggplant']
```

```
nItems = len(shoppingList)
```

```
myIndex = 0 # start with an index for the zero'th element
```

```
while myIndex < nItems:

    print(shoppingList[myIndex])

    myIndex = myIndex + 1 # increment the index
```

The idea here is to create a `myIndex` variable that starts at zero. Each time through the loop, we use that variable as an index, get the element at that position, and print it.

Then we increment the variable, preparing for the next time through the loop. The code produces the correct result but seems a little “clunky.” You have to remember a lot of details and get them all right to make this loop work correctly.

203

Chapter 7 Lists

for Statements and for Loops

The people who designed Python came up with a better way to handle iterating through a list. As long-time programmers, they noticed that this pattern of looping and doing something with each element of a list happens quite often. So they came up with an additional statement and a new type of loop that gives you an extremely simple way to iterate through a list. It is called the `for` statement. Here is the generic form: `for <elementVariable> in <list>: <indented statement(s)>`

The `for` statement is made up of new keyword, `for`, a variable name, another new keyword, `in`, and then the list you want to iterate through. The statement ends with a colon. After the colon is an indented block of `statement(s)` called the *body* of the loop.

Together, the `for` statement and the indented block are called a `for` loop.

The key to the `for` loop is the `<elementVariable>`. Here’s how it works. The `for` statement causes the body of the loop to execute once for every element in the `<list>`.

Each time through the loop, the variable you specify as `<elementVariable>` is set

to the value of the next element in the list.

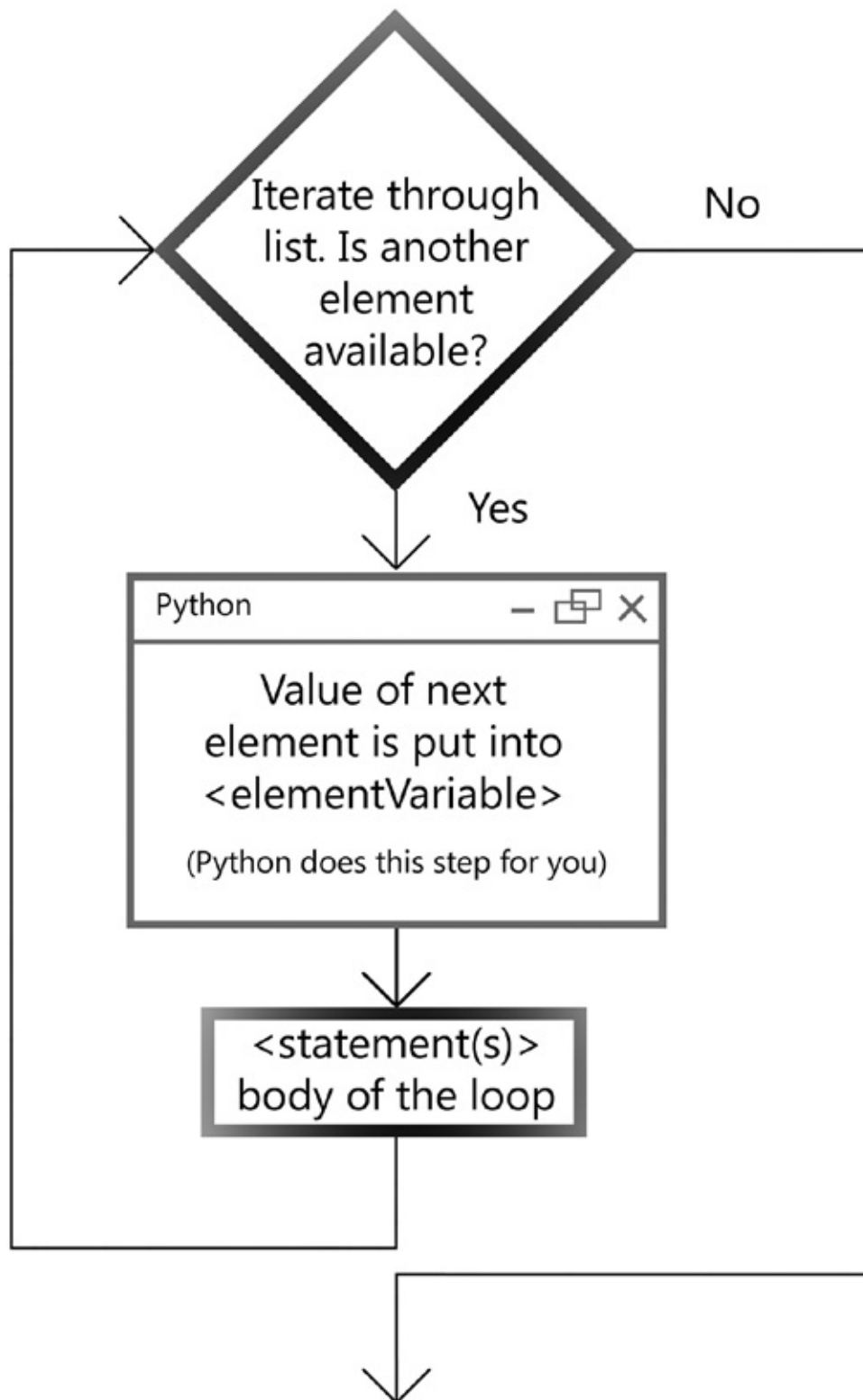
When you see a for statement, think of it as saying, “for each element in the list.”

Notice the new in keyword, which makes the for statement very English-like and readable. For example, let’s say you saw a for statement like this:

```
for name in namesList:
```

You could read it as, “for each name in the list namesList”—that is, it would iterate through namesList, and in each iteration would set the variable name to the next name in the namesList.

The flowchart of a for loop is shown in [Figure 7-1](#).



Chapter 7 Lists

Figure 7-1. The flowchart of a for loop

Notice that you do not need to use any index to get the value of each element in your list. The for loop does that for you automatically. It takes care of the tedious bookkeeping. This syntax is extremely elegant and simple.

Let's build a simple example. Suppose you want to print out your shopping list with one item per line:

```
shoppingList = ['apples', 'bananas', 'cherries', 'dates', 'eggplant']
```

```
for anItem in shoppingList:
```

```
    print(anItem)
```

205

Chapter 7 Lists

In this example, every time through the loop, the `anItem` variable is given the value of the next element in `shoppingList`, and the body of the loop prints each element on a separate line.

Here is another simple example. In this code, we have a list of teammates, and we want to say, "Good luck," to each of them: `teammatesList = ['Joe', 'Sue', 'Marcia', 'Sally']`

```
for teammate in teammatesList:
```

```
    print('Good Luck ' + teammate)
```

This code iterates through `teammatesList`. Each time through the list, the next element of `teammatesList` is put into the `teammate` variable. In the body of the loop, a separate greeting message is printed for each person: Good Luck Joe

Good Luck Sue

Good Luck Marcia

Good Luck Sally

>>>

Programming Challenge 2

In this challenge, you have an opportunity to use a for loop to iterate through a list.

Write a program that starts with a list of numbers. For example, write a list like this: `numbersList = [23, -10, 37, 4.5, 0, 123.4]`

Then use a for loop to add all the numbers in the list. Print the total.

Here is the solution:

```
# Calculate the total of numbers in a list
```

```
numbersList = [23, -10, 37, 4.5, 0, 123.4]
```

```
total = 0
```

```
for number in numbersList:
```

```
    total = total + number
```

```
print('The total of all numbers is:', total)
```

```
206
```

Chapter 7 Lists

The key to writing this code is to create a variable with a name like `total` and initialize it to zero. Then build a for loop to iterate through the list of numbers. Every time through the loop, the number variable is given the value of the next number in the list. In the body of the loop, add each value to the previous total. At the end, print the total: >>>

The total of all numbers is: 177.9

Generating a Range of Numbers

There are many situations where you would like to have a consecutive ordered sequence of integer numbers. For example, imagine you are doing some math, such as adding up the numbers from 1 to n . To help with problems like this, Python has a built-in function called `range` that creates an ordered sequence of integers. The typical way of calling the range function is like this:
`range(<lowEnd>, <upToButNotIncludingHighEnd>)`

The range function generates a collection of numbers that includes the `<lowEnd>` value and goes consecutively up to, but does not include, the value of `<upToButNotIncludingHighEnd>`. This approach to defining a range of numbers is identical to the way that `random.randrange` works.

Note Because

`range` is very often used with lists, you can call the range function

with a single argument instead of two arguments. When called this way, the single argument represents only the high end of the range. the low end of the range defaults to be zero. For example, specifying `range(25)` is identical to specifying `range(0, 25)`, and will both create a collection of numbers from 0 to 24. to make things explicitly clear, we always use the range function with two arguments.

The interesting use case for the range function is in a for statement. Remember that a for statement is designed to let you easily iterate through a collection, typically through a list. The range function works seamlessly in a for statement. Here is a good example: `for number in range(0, 10): print(number)`

207

Chapter 7 Lists

This for statement iterates through a collection of numbers from 0 to 9. Every time through the loop, it assigns the next value to the variable `number`. This code would print the numbers 0 to 9, each on a separate line.

Programming Challenge 3

This challenge gives you practice in using the range function in a for loop. The challenge is to write a program that allows the user to enter an integer. The program should calculate and print the total of all integers from 1 up to and including the user's number.

For example, if the user enters 3, then the program should add up $1 + 2 + 3$ and print the total of 6. If the user enters 10, then the program should calculate the total of $1 + 2 + 3 + \dots$

+ 10, and output a total of 55.

This is a solution to the challenge:

```
# Calculate the total of numbers up to a number entered by the user
```

```
usersNumber = input('Please enter an integer: ')
```

```
usersNumber = int(usersNumber)
```

```
highEndOfRange = usersNumber + 1
```

```
total = 0
```

```
for number in range(1, highEndOfRange):
```

```
total = total + number
```

```
print('The total numbers from 1 to', usersNumber, 'is', total)
```

This uses a similar approach to adding up a list of numbers as used in the previous coding challenge (by starting a total at zero and then adding each number). But in this challenge, the numbers to be added are not predefined. Instead, you must generate the numbers using a call to the range function. In order to get the proper upper bound for the call to range, you have to add one to the user's number, because the value of highEndOfRange is not included in the range itself. Here is the output of a few runs of this program: >>>

```
Please enter an integer: 3
```

The total numbers from 1 to 3 is 6

208

Chapter 7 Lists

```
>>> ===== RESTART
=====
```

```
>>>
```

Please enter an integer: 10

The total numbers from 1 to 10 is 55

```
>>> ===== RESTART
=====
```

```
>>>
```

Please enter an integer: 100

The total numbers from 1 to 100 is 5050

```
>>> ===== RESTART
=====
```

```
>>>
```

Please enter an integer: 1000

The total numbers from 1 to 1000 is 500500

```
>>>
```

Note in python 2, the range function worked differently. it created and returned a list. if the requested range was large, the list might have taken up a considerable amount of memory. in python 3, the range function does not return a list. instead, range has become a new type called a generator. a *generator* creates the next number in the sequence every time a new number is needed. therefore, when using range in python 3 in a for loop, there is a good chance that your loop may

run faster than the equivalent code in python 2. (the python 2 xrange function is the equivalent of the python 3 range function.) **Scientific Simulations**

In the scientific community, computers are often used to simulate the outcome of a large number of trials. In each trial, one or more pieces of data are given randomized values from all possible values. The scientists then look at the result of many trials to see if they can identify patterns.

209

		Die 2					
		1	2	3	4	5	6
Die 1	1	Doubles					
	2		Doubles				
	3			Doubles			
	4				Doubles		
	5					Doubles	
	6						Doubles

Chapter 7 Lists

Consider a simulation of rolling dice. In our simulation, we will perform many rounds of rolling two six-sided dice and then we will count the number of times that the dice generate *doubles* (that is, when both dice show the same value or *face*).

First, let's do a little math to see what we would expect for an answer. In each round, the first die can have any value from 1 to 6, and so can the second die. [Figure 7-2 is a](#)

chart showing all possible rolls of two dice.

Figure 7-2. *Possible rolls of two dice*

In this chart, the left side shows the possible faces for the first die. The top shows the possible faces for the second die. Out of the 36 possible combinations, 6 of them result in a doubles. That means that we should expect 6/36ths or 1/6th or 16.666666 percent of rolls to be doubles.

Here's the code for doing this simulation. We'll ask the user to enter the number of rounds, and for each round, we'll simulate rolling two dice: # Dice: count doubles in user-defined number of rounds

```
import random

# simulate rolling a six-sided die and return its value

def rollOneDie():

    # generate a random numbers between 1 and 6

    thisFace = random.randrange(1, 7)

    return thisFace

nDoubles = 0

n10
```

Chapter 7 Lists

```

maxRounds = input('How many rounds do you want to do? ')
maxRounds = int(maxRounds)

for roundNumber in range(0, maxRounds):

    die1 = rollOneDie()

    die2 = rollOneDie()

    if die1 == die2:

        nDoubles = nDoubles + 1

percent = (nDoubles * 100.0) / maxRounds

print('Out of', maxRounds, 'you rolled', nDoubles, 'doubles, or',
percent, '%')

```

In this program, the user specifies a number of rounds to roll two dice. As an example, let's say that the user wants to run ten rounds of dice rolls. We take the 10 the user gives us, convert it to an integer, and use it in a call to the range function. Passing 0 and 10 to range builds a collection of the numbers from 0 to 9. This is an example of zero-based counting; 0 through 9 is ten numbers, so the code will run through the loop ten times. The roundNumber variable is given the value of the next number in the collection, but we never use that variable anywhere in this loop. (If we wanted to report the result of every round, we could print that value each time through the loop.) The key concept here is that we are using the range function to help us go through the loop the correct number of times. In essence, the for loop is acting as a counter for us.

In each round (every time through the loop), we call the rollOneDie function twice.

rollOneDie does what its name implies and simulates the rolling of a single die. We assign the answers to two different variables: die1 and die2. If these two

we assign the answers to two different variables, die1 and die2. If these two variables have the same value, then we had a doubles, and we increment the count of doubles. When the loop is finished, we do a calculation of percentage (multiplying by 100.0 ensures that this will be a floating-point calculation), and we print the answer.

Here is the output of a sample run:

```
>>>
```

```
How many rounds do you want to do? 1000
```

```
Out of 1000 you rolled 158 doubles, or 15.8 %
```

```
>>>
```

```
211
```

Chapter 7 Lists

If we want to run our simulation again, we would have to run the program again.

Instead, let's make a modification to the code to allow the user to continue to enter different values for the number of rounds: # Dice - count doubles in user-defined number of rounds ... repeated

```
import random
```

```
# simulate rolling a six-sided die and return its value
```

```
def rollOneDie():
```

```
# generate a random numbers between 1 and 6
```

```
thisFace = random.randrange(1, 7)
```

```
return thisFace
```

```
while True:
```

```
nDoubles = 0
```

```

maxRounds = input('How many rounds do you want to do? (Or ENTER to
quit): ')

if maxRounds == "":
    break

try:
    maxRounds = int(maxRounds)
except:
    print('Please enter an integer number.')
    continue # go back to the while statement

for roundNumber in range(0, maxRounds):
    die1 = rollOneDie()
    die2 = rollOneDie()

    if die1 == die2:
        nDoubles = nDoubles + 1

    percent = (nDoubles * 100.0) / maxRounds

    print('Out of', maxRounds, 'you rolled', nDoubles, 'doubles, or',
percent, '%')

print('OK Bye')

```

212

Chapter 7 Lists

In this version, the code has been modified so that the main portion is now inside a larger while loop. Each time through the outer while loop, we ask the user how

many rounds they want to do. The program also has a try/except to ensure that the value the user enters is an integer. The program keeps running simulations until the user presses Enter (Windows) or Return (Mac) to exit.

Here is the output of a run where we entered increasingly larger values:

```
>>>
```

```
How many rounds do you want to do? (Or ENTER to quit): 1000
```

```
Out of 1000 you rolled 164 doubles, or 16.4 %
```

```
How many rounds do you want to do? (Or ENTER to quit): 10000
```

```
Out of 10000 you rolled 1690 doubles, or 16.9 %
```

```
How many rounds do you want to do? (Or ENTER to quit): 100000
```

```
Out of 100000 you rolled 16638 doubles, or 16.638 %
```

```
How many rounds do you want to do? (Or ENTER to quit): 1000000
```

```
Out of 1000000 you rolled 166751 doubles, or 16.6751 %
```

```
How many rounds do you want to do? (Or ENTER to quit): 10000000
```

```
Out of 10000000 you rolled 1666941 doubles, or 16.66941 %
```

```
How many rounds do you want to do? (Or ENTER to quit):
```

```
OK Bye
```

```
>>>
```

There are two interesting things to note here. First, these simulations run quite quickly. Even with the last one, where we did ten million rounds, the program took only a matter of seconds. Second, notice that the more rounds we ran, the more accurate the answer was—the closer it got to the predicted value of 16.666666.

Python is becoming more and more popular in the scientific community because

of these two reasons. Very often, scientists set up random simulations like this and then run them a large number of times to test out theories. Further, the random distribution of results is extremely even. The fact that we get a result very close to 16.666666

demonstrates this.

213

Chapter 7 Lists

List Manipulation

Let's go back to our example of a shopping list one more time, but this time consider what happens to a shopping list in a typical house. Right after a shopping trip, you might put up a new, empty shopping list on the refrigerator. As you notice that you are running low on groceries, you add items to the list. So maybe you add three items to your list one day, add two more the next day, and another the following day. Later, you move a box of cereal in your pantry and discover a hidden box of crackers that was on your list. You go back to the list and cross off crackers. If your list becomes long, you probably want to see if an item already appears in the list before adding it. You may also want to count the number of occurrences of an item to see if it appears more than once.

Python provides many built-in operations that allow you to manipulate and search through lists. The syntax of these operations is a little different from what we have seen before. This is the general syntax: `<listVariable>.<operation>` (`<any argument(s)>`)

THE “OBJECT” IN COMPUTER SCIENCE

in the world of computer science, there is an important concept called an *object*. My definition of an object is: *data—and code that acts on that data—over time*. although objects are beyond the scope of this book, i can tell you that internally in python, all lists are implemented as objects. the *data* (from my definition) is the content of the list—the collection of elements.

the *code* (from my definition) is the operations that act on any list. these list operations are available on any list just because they are lists. in this sense, each list object “knows” how to do each of these operations. Generically, the code of

every object is provided by functions, but these functions go by another name. When functions are applied to an object, they are called *methods of an object*. this is the syntax used to call a method of an object: <object>.<method>(<any argument(s)>)

that is why the syntax of the list operations in table [7-1 look the way they do.](#)

214

Chapter 7 Lists

Table 7-1. *The Built-In List Operations*

Operation	Description
<list>.append(<thing>)	add <thing> to the end of a list
<list>.count(<thing>)	returns the number of times <thing> was found in the <list>
<list>.	
	appends all elements in <otherList> to <list>
extend(<otherList>)	
<list>.index(<thing>)	returns the first index in the <list> where <thing> is found
<list>.insert(<thing>,	
	inserts <thing> into the <list> at position <index>
<index>)	
<list>.pop()	

`<list>.pop()`

remove and return the last element from a `<list>`

`<list>.pop(<index>)`

remove and return the element from a `<list>` at the given

`<index>`

`<list>.remove(<thing>)`

Find first occurrence of `<thing>` in a `<list>` and remove it

`<list>.reverse()`

reverse the position of all the elements in a `<list>`

`<list>.sort()`

sort elements in a `<list>` from low to high

The full documentation on all list operations can be found in the official Python documentation at <https://docs.python.org/3/tutorial/datastructures.html> in section 5.1.

The keyword `in` can also be used as an operator with a list:

`<value> in <listVariable>`

This syntax defines a Boolean expression that will generate a `True` if the value is found in the list, and a `False` if the value is not found. This type of expression can be used in an `if` statement or a `while` loop. The keywords `not in` can be used to reverse the result.

215

Chapter 7 Lists

List Manipulation Example: an Inventory Example

Consider an adventure game where you wander around a landscape. Games like

this often allow you to maintain an inventory. At the start of the game, you have nothing, or an empty inventory. As you move about the environment, you find different items and can add them to your inventory. Later in the game, you may find yourself in a situation where you need to use something in your inventory to get out of a tricky situation. Here is an example of some code that can simulate these actions—first, let’s build up an inventory from scratch: >>> inventoryList = [] # start as an empty list.

```
>>>
```

```
>>> inventoryList.append('treasure')
```

```
>>> print(inventoryList)
```

```
['treasure']
```

```
>>>
```

```
>>> inventoryList.append('magic stones')
```

```
>>> print(inventoryList)
```

```
['treasure', 'magic stones']
```

```
>>>
```

```
>>> inventoryList.append('potion')
```

```
>>> print(inventoryList)
```

```
['treasure', 'magic stones', 'potion']
```

```
>>>
```

We started with an empty list and as we found items, we appended them to the list.

Our list now has three elements. Later in the game, we learn that in order to kill a dragon, we need to throw some magic stones at it: >>> print('magic stones' in inventoryList)

```
True
```

```
True
>>>
>>> indexOfStones = inventoryList.index('magic stones')
>>> itemToThrow = inventoryList.pop(indexOfStones)
>>> print(inventoryList)
['treasure', 'potion']
>>> print(itemToThrow)
magic stones
>>>
```

216

Chapter 7 Lists

First, we check to ensure that we have the magic stones in our inventory by using the `in` operator. Seeing that we have the stones, we check to see where the magic stones live in our inventory by using the `index` operation. Once we have the index of where they are found, we use the `pop` operation to remove the magic stones from our inventory list and put them in a variable so that we can then throw the stones at the dragon.

Pizza Toppings Example

Let's wrap up this chapter by building a program that creates and modifies a list, one that uses many built-in list operations, while loops, and for loops.

In this sample program, you own a pizzeria. Customers are allowed to get any toppings on their pizza that they want. Your program needs to cater to their wishes. The program will handle the following operations:

- *a* or *add*: Adds a topping

- *c* or *change*: Changes a topping

- o or order: Orders the current pizza
- q or quit: Quits the program
- r or remove: Removes a topping
- s or startover: Starts the current pizza over

Here is the pseudocode of our program:

Function To Show Pizza Toppings

If there are no toppings, say there are none

Else

print each topping on a separate line

Print Welcome message, instructions, and large form of menu

Loop forever

Show short form of menu

Ask the user what they want to do:

If "add"

ask user what topping to add, add it

Else if "change"

217

Chapter 7 Lists

find topping to change

ask user what topping to change to, change it

Else if "order"

Show pizza being ordered

Thank user

Ask if they want to order another

If yes, start over

Else quit

Else if "remove"

Ask user what topping to remove

Remove that topping if found

Else if "startover"

Reset to starting state

Else

Tell user we did not understand

Show the current pizza

Based on that approach, we can write the code in Python. A key concept driving this program is that we will maintain the user's topping choices as a Python list. Each section of the code uses some different list operation to manipulate that list. This is the longest program we have seen so far. If you read through the code slowly to see how it matches the pseudocode, it should not be too hard to follow:

```
# Pizza toppings program
```

```
# Function to show the list of toppings
```

```
def showPizzaToppings(theList):
```

```
    print()
```

```
    if len(theList) == 0:
```

```
        print('Your pizza has no toppings.')
```

```
print('Your pizza has no toppings.')
```

```
else:
```

```
print('The toppings on your pizza are:')
```

```
print()
```

```
for thisItem in theList: # iterate through the list, print each item
```

```
print(' ' + thisItem)
```

```
print() # blank line
```

```
218
```

Chapter 7 Lists

```
# main code
```

```
print('Welcome to my Pizzeria, where you get to choose your toppings.')
```

```
print('When prompted, enter the first letter or the full word of what you  
want to do.')
```

```
print()
```

```
print('---- Operations ----')
```

```
print('a/add Add a topping')
```

```
print('c/change Change a topping')
```

```
print('o/order Order the pizza')
```

```
print('q/quit Quit')
```

```
print('r/remove Remove a topping')
```

```
print('s/startover Start over')
```

```
... ^
```

```

print()

toppingsList = [ ] # begin as an empty list

while True:

    print('What would you like to do?')

    menuChoice = input(' add, change, order, quit, remove, startover: ')

    if (menuChoice == 'a') or (menuChoice == 'add'): # add a topping

        newTopping = input('Type in a topping to add: ')

        toppingsList.append(newTopping) # append adds to the end of a list

    elif (menuChoice == 'c') or (menuChoice == 'change'): # change a
topping

        oldTopping = input('What topping would you like to change: ')

        if oldTopping in toppingsList: # is it in the list

            index = toppingsList.index(oldTopping) # find out where it is
in the list

            newTopping = input('What is the new topping: ')

            toppingsList[index] = newTopping # set a new value at that index

        else:

            print(oldTopping, 'was not found.')

```

219

Chapter 7 Lists

```

elif (menuChoice == 'o') or (menuChoice == 'order'): # order the pizza

```

```
showPizzaToppings(toppingsList)

print()

print('Thanks for your order!')

print()

another = input('Would you like to order another pizza (y/n) ? ')

if another == 'y':

    toppingsList = [ ] # reset to the empty list

else:

    break

elif (menuChoice == 'q') or (menuChoice == 'quit'): # quit

    break

elif (menuChoice == 'r') or (menuChoice == 'remove'): # remove a

    topping

    delTopping = input('What topping would you like to remove: ')

    if delTopping in toppingsList: # check to see if the topping is in

        our list

        index = toppingsList.index(delTopping) # find out where it is

        toppingsList.pop(index) # remove it

        # The code above only removes the first occurrence of the

        topping.

    else:
```

```

print(delTopping, 'was not found')

elif (menuChoice == 's') or (menuChoice == 'startover'): # reset to no
toppings

print("OK, let's start over.")

toppingsList = [ ] # reset to the empty list

else:

print("Uh ... sorry, I'm not sure what you said, please try again.")

showPizzaToppings(toppingsList) # show the list of toppings on the pizza

print()

print('Goodbye')

```

220

Chapter 7 Lists

The key to this program is the toppingsList list variable in the main section of the code. It starts off as the empty list to represent a pizza with no toppings on it. The user can then add toppings to the pizza, and in response, the program uses the append operation to add to the end of the list. For a change operation, we first use the in operator to ensure that the topping to be changed exists in the list of toppings. If so, we replace the old topping with the new topping by using the index of where the old topping was found. Ordering winds up resetting the toppingsList back to the empty list. Should the user ask to remove a topping, the program checks to see if that topping is in the list, and if so, finds the index of the topping and uses the pop operation to remove that topping. Starting over simply resets to the empty list.

At the top of the program is a small function that prints the list of toppings. If there are none, the function prints that. Otherwise, it uses a for loop to iterate through the list of the pizza toppings and prints each one on a separate line. The output of a typical run could look like this: Welcome to my Pizzeria, where you get to choose your toppings.

When prompted, enter the first letter or the full word what you want to do.

---- Operations ----

a/add Add a topping

c/change Change a topping

o/order Order the pizza

q/quit Quit

r/remove Remove a topping

s/startover Start over

What would you like to do?

add, change, order, quit, remove, startover: add

Type in a topping to add: mushrooms

The toppings on your pizza are:

mushrooms

What would you like to do?

add, change, order, quit, remove, startover: a

Type in a topping to add: pineapples

The toppings on your pizza are:

221

Chapter 7 Lists

mushrooms

pineapples

What would you like to do?

add, change, order, quit, remove, startover: uvwxyz

Uh ... sorry, I'm not sure what you said, please try again.

The toppings on your pizza are:

mushrooms

pineapples

What would you like to do?

add, change, order, quit, remove, startover: add

Type in a topping to add: bacon

The toppings on your pizza are:

mushrooms

pineapples

bacon

What would you like to do?

add, change, order, quit, remove, startover: change

What topping would you like to change: bacon

What is the new topping: pepperoni

The toppings on your pizza are:

mushrooms

pineapples

pepperoni

pepperoni

What would you like to do?

add, change, order, quit, remove, startover: r

What topping would you like to remove: pineapples

The toppings on your pizza are:

mushrooms

pepperoni

222

Chapter 7 Lists

What would you like to do?

add, change, order, quit, remove, startover: o

The toppings on your pizza are:

mushrooms

pepperoni

Thanks for your order!

Would you like to order another pizza (y/n) ? n

Goodbye

>>>

Summary

In this chapter, you learned how to store, access, retrieve, and manipulate ordered collections of data called *lists*. You learned that lists are made up of elements, and each element has a position known as its *index*. Lists are defined in Python using the square brackets with elements separated by commas. We can

refer to an individual element in a list by using the bracket syntax and specifying the index of the element we want. An index can be a constant, a variable, or an expression.

We built a fun Mad Libs game and then modified it to use lists. The program chose random words from a number of lists. We used the `len` function to find out how many elements are in a list. You saw how to use a list as an argument in a function call.

Then we explored the topic of iteration—the ability to visit all elements of a list. To do this, we used a `for` statement and built a `for` loop. We used iteration to sum up the numbers in a list. We found that the `range` function can be used to generate a list of consecutive integers, and is often used in `for` loops. We demonstrated how the `range` function can be used to run a loop through a set number of iterations.

Finally, we introduced a number of list-manipulation operations that can be used to modify and search through the contents of a list. We ended with a demonstration program that maintains a list of pizza toppings as a list and uses these list manipulation operations to do so.

223

CHAPTER 8

Strings

Other than using them to nicely format output, we haven't talked much about strings. In this chapter and the next two chapters, we get heavily into strings. I'll show you how to manipulate them and find smaller strings within larger strings.

We started using strings in [Chapter 1](#) with this statement:

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

Later, we talked about how you get input from the user as a string, and how to convert that input into a number:

```
>>>
```

```
>>> age = input('Please enter your age: ')
```

Please enter your age: 24

```
>>> age = int(age)
```

```
>>>
```

Then I showed you how to concatenate strings, like this:

```
>>>
```

```
>>> string1 = 'Hello'
```

```
>>> string2 = 'there'
```

```
>>> greeting = string1 + ' ' + string2
```

```
>>> print(greeting)
```

Hello there

```
>>>
```

This chapter covers the following topics:

- len function applied to strings
- Indexing characters in a string
- Accessing characters in a string

225

© Irv Kalb 2018

I. Kalb, *Learn to Program with Python 3*, https://doi.org/10.1007/978-1-4842-3879-0_8

Chapter 8 StringS

- Iterating through characters in a string
- Creating a substring: a slice
- Programming challenge 1: creating a slice
- Additional slicing syntax
- Slicing as applied to a list
- Strings are not changeable
- Programming challenge 2: searching a string
- Built-in string operations
- Examples of string operations
- Programming challenge 3: directory style

len Function Applied to Strings

Though it may not seem obvious, strings are very similar to lists. Think of a string as a list of characters. That is worth repeating: *think of a string as a list of characters*. Many of the operations you can do with lists, you can also do with strings.

For example, like a list, a string can be any length. To find out how many elements are in a list, you use the len built-in function. But len can also be used on a string: >>> state = 'Mississippi'

```
>>> theLength = len(state)
```

```
>>> print(theLength)
```

```
11
```

```
>>>
```

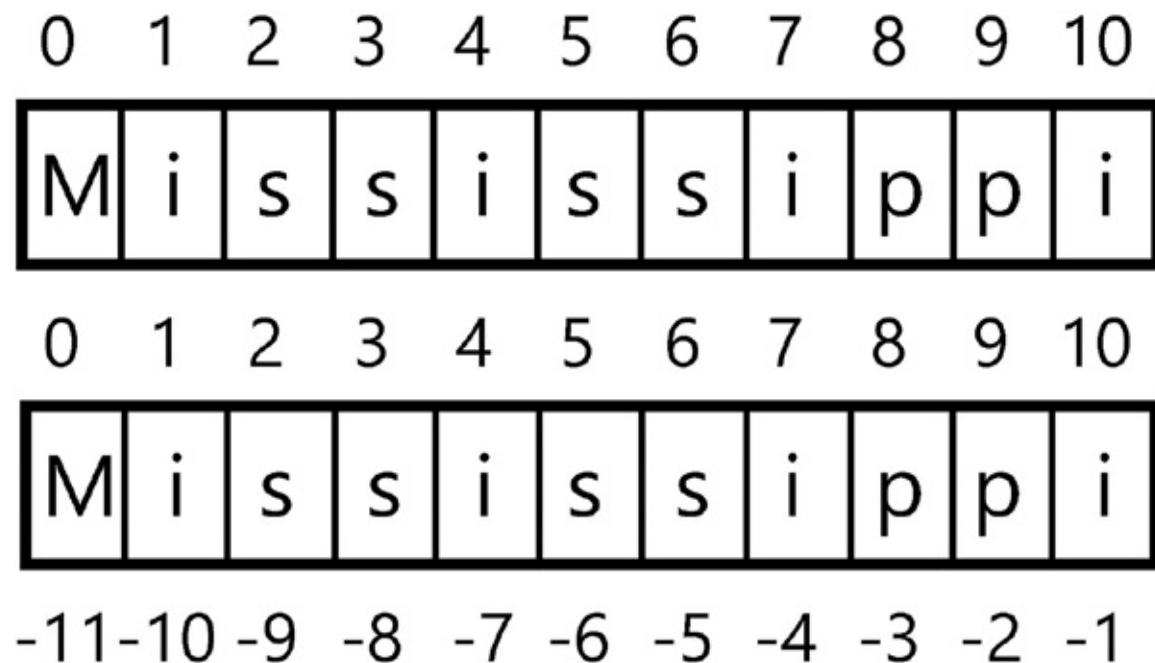
Indexing Characters in a String

Again, if we think of a string as a *list* of characters, then we can think of each character as an *element*. Further, we can use an index to refer to a character in a string the same way we index an element in an array. Remember from the definition, the index is the position of an element. With respect to a string, an index is the position of a character in a string.

Given the earlier assignment statement, where we set the variable state to the string

'Mississippi', Figure [8-1](#) shows the indices of the characters in the string.

226



Chapter 8 StringS

Figure 8-1. The indices of the characters in the string

The string in Figure 8-1 has 11 characters. Notice that the characters in a string are numbered (or indexed) identically to the elements in a list. What we humans would think of as the first character (uppercase *M*) in Python is considered the character at index 0.

The last character is always found at an index equal to the length of the string minus one.

Because there are 11 characters in this string, the last character is found at index 10.

Similar to the indices of a list, you can also use negative indices to access the characters in a string, as shown in Figure [8-2](#).

Figure 8-2. Negative indices of the characters in a string

You can think of the negative index as the positive index minus the length of the string. For example, the first *p* in Mississippi is at index 8. But it can also be addressed by using `-3`, because $8 - 11 = -3$. In practice, negative indexing is not used very often.

Perhaps its most useful purpose is when you want to get the last character in a string; its index is always `-1`.

Accessing Characters in a String

As with a list, we can also use the bracket syntax to identify a character at a specific index in a string: `>>> print(state[0])`

`M`

`>>> print(state[1])`

`227`

Chapter 8 StringS

i

```
>>> print(state[2])
```

s

```
>>>
```

If you try to access a character that is beyond the end of a string, Python will generate an appropriate error message: `>>> print(state[1000])` # only has 11 characters

Traceback (most recent call last):

File "<pyshell#12>", line 1, in <module>

```
print(state[1000])
```

IndexError: string index out of range

```
>>>
```

Remember that there is the special case of the *empty string*—a string with no characters in it. Its length is zero, and indices do not apply: `>>> myString = "`

```
>>> print(len(myString))
```

0

```
>>>
```

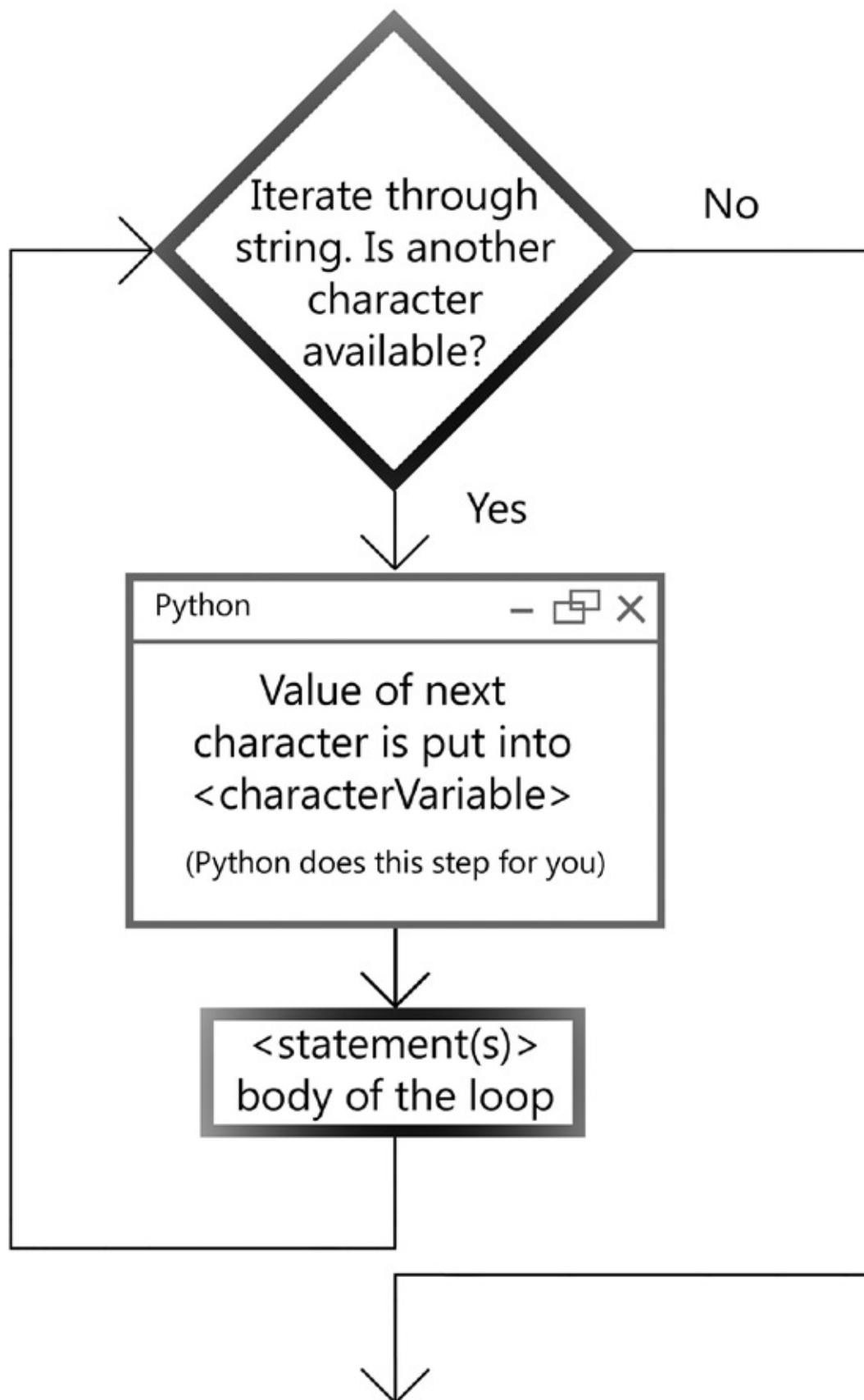
Iterating Through Characters in a String

Similar to the way we iterate through all elements of a list, we often want to iterate through all characters in a string. And similar to the while loop that I first showed to iterate through a list, we could build an identical while loop to iterate through a string: # Iterate through a string

This is the WRONG approach, just showing a concept!

```
state = 'Mississippi'
myIndex = 0
while myIndex < len(state):
    print(state[myIndex])
    myIndex = myIndex + 1
```

228



Chapter 8 StringS

This code would correctly print all characters in the string, one per line. But just like when visiting all elements in a list, the for statement allows you to easily loop through (or iterate through) all characters in a string. [Figure 8-3 is the same](#) flowchart of a for loop that we saw earlier, but this time applied to iterating through the characters in a string.

Figure 8-3. *The flowchart of a for loop iterating through a string* 229

Chapter 8 StringS

This syntax and the operation are identical to the for loop used to iterate through a list:

```
for <characterVariable> in <string>:
```

```
<indented statement(s)>
```

The only difference is that because we are iterating through a string, the

<characterVariable> is given the next character in the string (rather than the next element in a list). For example, this myString = 'abcdefg'

```
for letter in myString:
```

```
    print(letter)
```

prints this:

a

b

c

d

e

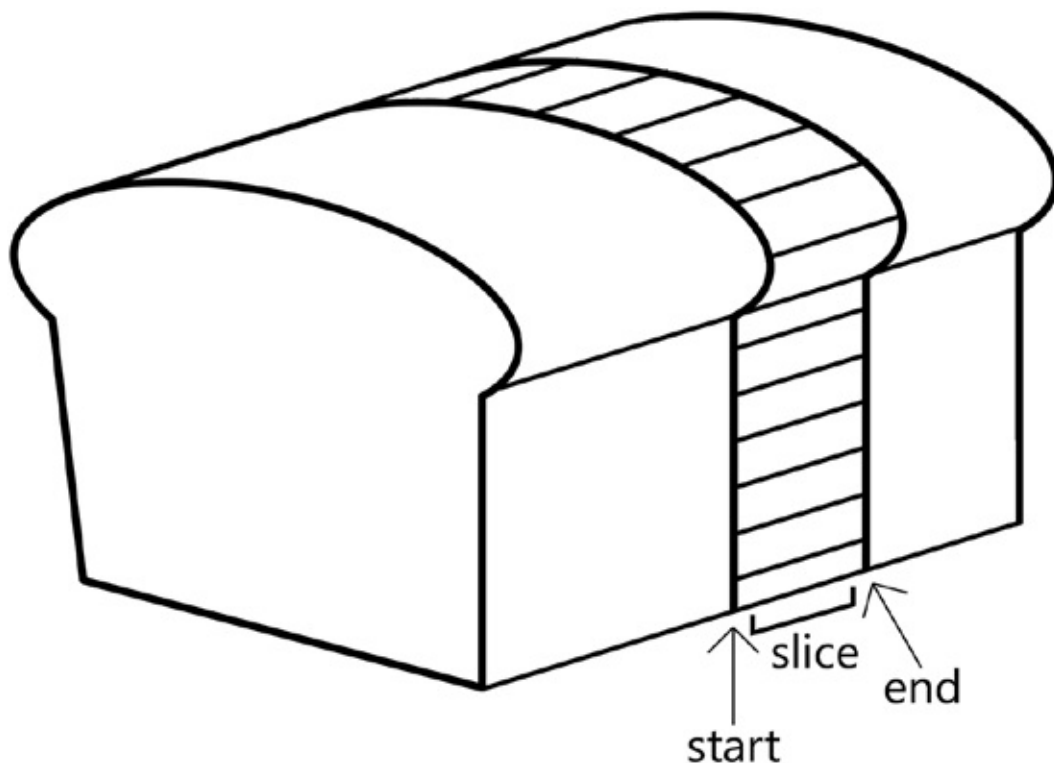
f

g

Creating a Substring: A Slice

Very often when you are dealing with strings, you want to extract a shorter string from a longer string—for example, when you want to find a particular piece of information within a string. In programming, this is generally called *creating a substring*. In Python, we do this by taking a *slice* of a string, and in the Python world, this is commonly called *slicing*. Figure [8-4](#) illustrates this.

230



Chapter 8 StringS

Figure 8-4. *Think of a string as a loaf of bread and take a slice of it* To make a slice of a string, we have to specify the start index and the end index of the slice we want to create. If you think of a string as a loaf of bread, then the analogy of cutting a slice makes this concept very clear. But when you make a slice in a string, Python makes a copy of the characters in the slice. It does not remove those characters from the string.

To specify a slice, Python provides the following syntax:

```
<string>[<startIndex> : <upToButNotIncludingEndingIndex>]
```

Once again, we see this consistent concept of a Python range, where the start value is included and an ending value is not included. The character at

`<startIndex>` is included as the first character of the substring. But the ending index, `<upToButNotIncludingEndingIndex>`, is the index of the first character that is *not* included in our slice. For example, we could have a string like this:
`myName = 'Joe Schmoe'`

To get just the first name, we want to take a slice starting at index 0 (the *J*), through index 2 (the *e*). Therefore, to create a substring that includes just the first name, we would ask for this slice: `>>> print(myName[0:3])`

```
Joe
```

```
>>>
```

```
231
```

Chapter 8 StringS

To get the last name, we would use this slice:

```
>>> nChars = len(myName)
```

```
>>> print(nChars)
```

10

```
>>> print(myName[4 : nChars])
```

```
Schmoe
```

```
>>>
```

Notice that you can use constants, variables, or expressions in defining the starting or ending value of a slice.

Programming Challenge 1: Creating a Slice

To see if the concept of a slice is making sense, it's time for a programming challenge. In this challenge, we start with the following string: months = 'JanFebMarAprMayJunJulAugSepOctNovDec'

Your job is to write a program that allows the user to enter a month number and print the three-letter abbreviation for that month. For example, if the user enters **1**, the program should print Jan. If the user enters **12**, the program should print Dec.

To get you going in the right direction, I'll give you the “scaffolding” of this assignment and leave the tricky part for you: # Given a month number, find the three letter abbreviation for that month

```
months = 'JanFebMarAprMayJunJulAugSepOctNovDec'
```

```
monthNumber = input('Enter a month number (1-12): ')
```

```
monthNumber = int(monthNumber)
```

```
# Some code that generates the appropriate start and end indices.
```

```
# Generate the appropriate slice
```

```
monthAbbrev = months[startIndex : endIndex]
```

```
print(monthAbbrev)
```

```
months = 'JanFebMarAprMayJunJulAugSepOctNovDec'
```

This program can be done in two or three lines of code. (Do not cheat by using 12 if statements!).

232

Chapter 8 Strings

Hint think about the slice indices that have to be created for Jan and the slice that has to be created for Feb, for Mar, and for Apr. is there a pattern? Can you write some simple code that generates a solution to this pattern?

To work through a solution, let's start with a chart (see [Table 8-1](#)) that shows the mapping between the month number and the start index in our months string.

Table 8-1. Mapping Month Number to the Related Start Index

monthNumber

startIndex

1

Jan

0

2

Feb

3

3

Mar

6

4

Apr

9

...

12

Dec

33

The idea is to look for a pattern. If you look at these numbers long enough, you start to recognize that the start index can be calculated by taking the month number, subtracting one from it, and then multiplying the result by three.

Writing this in Python looks like this: `startIndex = (monthNumber - 1) * 3`

We can calculate the end index two different ways. We could look in our string and see which character is the end character for each month. (Remember, when making a slice, the end index is the index of the first character that is *not* included.) We could extend the chart as shown in [Table 8-2](#).

233

Chapter 8 StringS

Table 8-2. *Mapping Month Number to the Related Start and End Index*
monthNumber

startIndex

endIndex

1

Jan

0

3

2

...

Feb

3

6

3

Mar

6

9

4

Apr

9

12

...

12

Dec

33

36

The values in the last column also make logical sense. Because each abbreviation is three letters long, it may now seem obvious that the end index is three more than each start index. Let's write the Python code for that: $\text{endIndex} = \text{startIndex} + 3$

Now we plug those two lines into our program, and the full solution becomes the following:

```
# Given a month number, find the three letter abbreviation for that month
```

```
months = 'JanFebMarAprMayJunJulAugSepOctNovDec'
```

```
monthNumber = input('Enter a month number (1-12): ')
```

```
monthNumber = int(monthNumber)
```

```
startIndex = (monthNumber - 1) * 3
```

```
endIndex = startIndex + 3
```

```
# Generate the appropriate slice
```

```
monthAbbrev = months[startIndex : endIndex]
```

```
print(monthAbbrev)
```

Here are a few runs to test our code:

```
>>>
```

```
Enter a month number (1-12): 1
```

```
Jan
```

```
234
```

```
Chapter 8 StringS
```

```
>>> ===== RESTART
=====
```

```
>>>
```

```
Enter a month number (1-12): 2
```

```
Feb
```

```
>>> ===== RESTART
=====
```

```
>>>
```

Enter a month number (1-12): 12

Dec

```
>>>
```

Additional Slicing Syntax

Python also allows additional syntax if you want the slice to start at the first character or end at the last character of a string. You can leave off the starting index of a string like this: `<someString>[: <upToButNotIncludingIndex>]`

This means to create a slice starting at the first character of a given string, and go up to but don't include the character at index `<upToButNotIncludingIndex>`. Similarly, you can also use this syntax: `<someString>[<startIndex> :]`

This means start at the given index and include all characters through the end of the string.

Finally, you can use this syntax:

```
<someString>[:]
```

That means to make a copy of the whole string. Here are some examples:

```
>>> sample = 'This is a sample string'
```

```
>>> print(sample[10:])
```

sample string

```
>>> print(sample[:16])
```

This is a sample

```
>>> print(sample[:])
```

This is a sample string

```
>>>
```

Slicing as Applied to a List

I didn't mention this in the last chapter, but the slicing syntax I have just shown for creating a substring can also be used with a list to create a sublist. The exact same syntax is used: `<someList>[<startingIndex> : <upToButNotIncludingIndex>]`

For example:

```
>>> startingList = [22, 104, 55, 37, -100, 12, 25]
```

```
>>> mySubList = startingList[3 : 6]
```

```
>>> print(mySubList)
```

```
[37, -100, 12]
```

```
>>>
```

Strings Are Not Changeable

There is one big difference between lists and strings. Strings are *not* changeable.

Remember that I said that lists are changeable, or *mutable*. In Python terms, strings are *immutable*. You cannot set or change an individual character in a string. For example, let's say you wanted to change a specific character of a string to some other character.

Let's try to change the second character of a given string to a different letter:

```
>>> someString = 'abcdefghijkl'
```

```
>>> someString[2] = 'x'
```

Traceback (most recent call last):

File "<pyshell#18>", line 1, in <module>

```
someString[2] = 'x'
```

TypeError: 'str' object does not support item assignment

```
>>>
```

This error message happens because strings are not changeable. However, you can

always create a new string or reassign a different value to an existing string variable. To change the second character of a string to another value, you have to reassign a string or create a new string. To accomplish our task, we can take this approach: 236

Chapter 8 Strings

```
>>> someString= 'abcdefghijkl'
```

```
>>> someString = someString[:2] + 'x' + someString[3:]
```

```
>>> print(someString)
```

```
abxdefghijkl
```

```
>>>
```

We've taken our original string, created a slice before the character we want,

concatenated the letter we want, and then concatenated another slice starting right after the character we wanted to eliminate. Finally, we assigned the resulting string back into our string variable.

Programming Challenge 2: Searching a String

In this challenge, I ask you to write a small function called `countCharInString`. It is passed the following two parameters:

- `findChar`: A character to find

- `targetString`: A string to be searched

It should return the number of times `findChar` is found in `targetString`.

```
"""
    countCharInString: A function that takes a character and a string as input and returns the number of times the character is found in the string.
    """
```

You can test your function with the following calls:

```
print(countCharInString ('s', 'Mississippi')) # expect 4
```

```
print(countCharInString ('p', 'Mississippi')) # expect 2
```

```
print(countCharInString ('q', 'Mississippi')) # expect 0
```

Here is a solution:

```
# Count a single char in another string
```

```
def countCharInString(findChar, targetString):
```

```
    count = 0
```

```
    for letter in targetString: # for each letter in the target string
```

```
        if findChar == letter: # if there is a match
```

```
            count = count + 1 # increment the count
```

```
    return count
```

```
print(countCharInString ('s', 'Mississippi')) # expect 4
```

```
print(countCharInString ('p', 'Mississippi')) # expect 2
```

```
print(countCharInString ('q', 'Mississippi')) # expect 0
```

```
print(countCharInString ('q', 'Mississippi')) # expect 0
```

```
print(countCharInString ('q', 'Mississippi')) # expect 0
```

And the following output is what we expect:

```
>>>
```

```
4
```

```
2
```

```
-
```

0

>>>

Built-in String Operations

Although it is fun to build these types of functions, it turns out we don't have to. The people who built Python have done all this work for us. In fact, there is a whole set of string manipulation routines built in to Python.

Similar to our discussion of lists in [Chapter 7](#), strings are also internally implemented as objects in Python. Because of that, you use the same syntax we used for list operations to invoke a string operation: `<string>.<operationName>(<optionalArguments>)`

Table [8-3](#) describes the most commonly used built-in string operations.

238

Chapter 8 StringS

Table 8-3. *The Most Commonly Used Built-In String Operations*

Operation	Description
<code><string>.count(<substring>)</code>	returns the number of times <code><subString></code> was found in <code><string></code> .
<code><string>.find(<subString>)</code>	returns the index of the first occurrence of <code><substring></code> in <code><string></code> . returns <code>-1</code> if <code><substring></code> is not found.
<code><string>.index(<subString>)</code>	

`<string>.index(<substring> ,`

returns the index of the first occurrence of

`<substring>` in `<string>`.

`<string>.lower()`

returns a lowercase version of `<string>`.

`<string>.lstrip()`

returns the string with leading (left) whitespace

removed.

`<string>.replace(<old>, <new>)`

returns a version of `<string>` where all `<old>` are

replaced by `<new>`.

`<string>.rstrip()`

returns the string with trailing (right) whitespace removed.

`<string>.startswith(<prefix>)`

returns True if `<string>` starts with `<prefix>`,

otherwise returns False.

`<string>.strip()`

returns the string with leading and trailing whitespace

removed.

`<string>.title()`

returns a version of `<string>` where the first letter

of every word is uppercase, and all other letters are

of every word is uppercase, and all other letters are lowercase.

```
<string>.upper()
```

returns an uppercase version of <string>.

To see the list of all string operations, you can enter this in the Shell:

```
dir('abc')
```

That prints out a list of the names of all string operations. You can ignore the ones that start with one or two underscores. The ones that seem human readable (at the end of the list) are the interesting ones. The full documentation on all string operations can be found in the official Python documentation at [https://docs.python.org/2/](https://docs.python.org/2/library/stdtypes.html)

[library/stdtypes.html](https://docs.python.org/2/library/stdtypes.html) in section 5.6.1.

239

Chapter 8 StringS

Examples of String Operations

In one of our earlier programming challenges, I asked you to write a function to count the number of times a character appears in a target string. Although it's good practice to write functions like this, built-in string operations are available to do much of the work like that for you. For example, the count operation does everything that our function does and more. The count operation finds not only a single character within another string, it finds a substring of any length in a string: >>> myString = 'Ask not what your country can do for you, ask what you can do for your country.'

```
>>> print(myString.count('o')) # how many of the letter o
```

11

```
>>> print(myString.count('can do'))
```

2

-

```
>>>
```

Whenever you ask the user for a text-based answer to a question, you can never know whether the user will enter the answer in all lowercase, all uppercase, or some mix of cases. This is a problem because Python string comparisons are case sensitive.

An answer of **OK** is not the same as **ok** and is not the same as **Ok**. Therefore, whenever you want to check for a user's text response, it is a good idea to convert the user's answer using either the lower operation (generally preferred) or the upper operation, before comparing their input. For example: >>>

```
userAnswer = input('Type OK if you want to continue: ')
```

Type OK if you want to continue: OK

```
>>> if userAnswer.lower() == 'ok':
```

```
# user answered OK, do whatever you need to do to continue
```

Another example is where you ask the user a yes-or-no question. Again, you cannot know in advance whether the user will type **Yes**, or **yes**, or **yES**, or even just the letter **y**.

You can use two string operations to handle all of these cases easily:

```
>>> userInput = input('Type yes to continue, no to quit: ')
```

Type yes to continue, no to quit: yes

```
>>> userInput = userInput.lower()
```

```
>>> if userInput.startswith('y'):
```

```
# user said yes, continue on with the program.
```

240

Chapter 8 StringS

In this example, we take whatever the user types and convert it to lowercase. Then we only look at the first character to see if the user's answer starts with the letter y.

Programming Challenge 3: Directory Style

It's time for the final programming challenge in this chapter. In this challenge, you ask the user to enter their name in the normal first name/last name style. Your job is to convert the name to directory style. Here are the details: 1. Ask the user to enter their name in the form <firstName><space>

<lastName>.

2. Take the name the user enters and find the index of the space.

3. Given that index, break up the user's string into a first name and last name.

4. Create a new string by reassembling the name to be shown in directory style, <lastName>, <space><lastName>, and print it.

Here is the solution:

```
# First name last name, produce directory style:
```

```
fullName = input('Please enter your full name: ')
```

```
indexOfSpace = fullName.index(' ')
```

```
firstName = fullName[:indexOfSpace]
```

```
lastName = fullName[indexOfSpace + 1:]
```

```
print(lastName + ', ' + firstName)
```

This one is fairly straightforward. The key is to find the index of the space. Once you find where the space character is in the string, you can use the slicing syntax to create a slice for the first name (starting at the first character), and a slice for the last name (that goes through the last character).

Often in programming, in order to eliminate potential errors where the program might crash, we use *defensive coding* techniques to ensure that the user provided valid input. In this programming challenge, our original solution assumed that the user entered a single space character in between the names. But what if the user forgot to enter a space, or entered multiple spaces, or entered spaces at the beginning and/or ending of the name? We can check for all these cases without crashing.

241

Chapter 8 StringS

Here is another version of the code that has some additional defensive coding to ensure that the program would not crash from these types of errors: # Read in first name last name, produce directory style with error

detection

while True:

fullName = input('Please enter your full name: ')

fullName = fullName.strip() # remove any spaces before or after

nSpaces = fullName.count(' ')

if nSpaces == 1: # OK if there is a single space

break

print('Please try again. Enter your name as first name, space, last name')

print()

indexOfSpace = fullName.index(' ')

firstName = fullName[:indexOfSpace]

firstName = firstName[0].upper() + firstName[1:] # Force first letter to

```
uppercase
```

```
lastName = fullName[indexOfSpace + 1:]
```

```
lastName = lastName[0].upper() + lastName[1:] # Force first letter to
```

```
uppercase
```

```
print(lastName + ', ' + firstName)
```

In addition to checking for incorrect spacing, the code also makes sure that the first letter of the first and last names are uppercase. The following is a sample run, first with an error and then with the user entering the name in all lowercase:

```
>>>
```

```
Please enter your full name: joeschmoe
```

```
Please try again. Enter your name as first name, space, last name
```

```
Please enter your full name: joe schmoe
```

```
Schmoe, Joe
```

```
>>>
```

```
242
```

Chapter 8 StringS

Summary

In this chapter, you learned to think of a string as a list of characters. You saw how the `len` function, indexing, and accessing characters in a string are identical to the way we use them in lists. You also learned that a `for` loop can be used to iterate through all the characters of a string. A new concept of a substring, known as a *slice* in Python, can be created using a new bracketing syntax.

You saw that the main difference between a string and a list is that a string is immutable (not changeable), whereas lists are mutable (easily changeable). I introduced a number of built-in string operations that can be used to manipulate string data.

The next two chapters continue our discussion of strings, with more examples of how strings are used in real-world programs.

243

CHAPTER 9

File Input/Output

In every program we have talked about so far, when the program ends, the computer forgets everything that happened in the program. Every variable you created, every string you used, every Boolean, every list—it's all gone. But what if you want to keep some of the data you generate in a program and save it for when you run the program later? Or maybe you want to save some data so that a different program could use the data you generated.

If you want to keep some information around between runs of a program, you need

a way of having what is called *persistent* data—you want the data to remain available on the computer. To do that, you need the ability to write to and read from a file on the computer.

This chapter discusses file input/output, often shortened to *file I/O*, and covers the following topics:

- Saving files on a computer

- Defining a path to a file
- Reading from and writing to a file
- File handle
- The Python `os` package
- Building reusable file I/O functions
- Example using our file I/O functions
- Importing our own modules
- Saving data to a file and reading it back

- Building an adding game
- Programming challenge 1

245

© Irv Kalb 2018

I. Kalb, *Learn to Program with Python 3*, https://doi.org/10.1007/978-1-4842-3879-0_9

Chapter 9 File input/Output

- Programming challenge 2
- Writing/reading one piece of data to and from a file
- Writing/reading multiple pieces of data to and from a file
- The join function
- The split function
- Final version of the adding game
- Writing and reading a line at a time with a file
- Example: multiple-choice test
- A compiled version of a module

Saving Files on a Computer

There are many examples of storing data in a file that you are already familiar with.

Think about a word processor or spreadsheet program. You create a document in your word processor or spreadsheet application, save it as a file on your computer, and then quit the program. Later, you reopen the word processor or spreadsheet program, reopen the saved file, and all the information you entered is brought back.

In fact, the Python source files that you write work the same way. You open the Python IDLE editor and create a Python source file (a document). As you edit your source code in IDLE, the content is kept in the memory of the computer. When you save it, the content (which is really just a string of text) is written to a file on the computer. You can then quit IDLE. Later, when you come back into IDLE and open the file, the text of your program is read in and displayed. IDLE displays the text, character by character, across each line.

Whenever it finds an end-of-line character, it moves down to the first character of the next line. The content is displayed for you, and you can edit again. Whenever you save, the current version of your program is written out to the file—again, as a long string of text.

But as I said at the start of this chapter, when we run a Python program and then stop it or quit IDLE, any data that we have manipulated in the program goes away. In order to save data, we need a way to write data from a running program to a text file, and when we run the program again, be able to read that data back into our program. Python allows programmers to easily write and read files of text. When dealing with files that contain only text, the convention is to name such a file with a .txt extension.

246

Chapter 9 File input/Output

Defining a Path to a File

When you want to read or write a text file, you must first identify which file you want to write to or read from.

Definition a

path is a string that uniquely identifies a file on a computer. (it is

sometimes called a *filespec*, short for *file specification*.) A path is a string. There are two different ways to specify a path: absolute and relative. An *absolute* path is one that starts at the top of the file system on your computer and ends in the name of the file. For example, an absolute path might look like this in Windows: C:\MyFolder\MySubFolder\MySubSubFolder\MyFile.txt

It might look like this on a Mac:

it might look like this on a Mac:

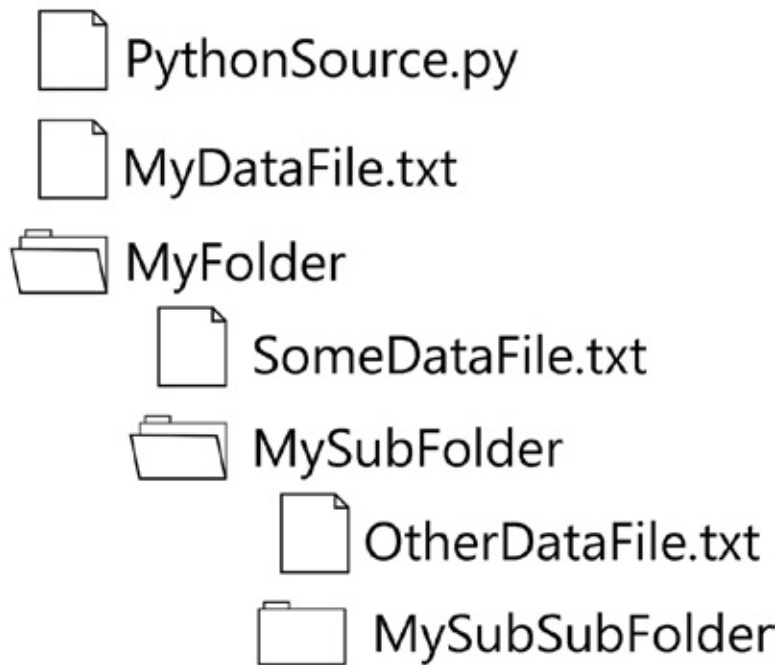
Macintosh HD/MyFolder/MySubFolder/MySubSubFolderMyFile/MyFile.txt

However, one of the great things about Python is that it is designed to allow programmers to write portable code, which can be used on different computers and on different operating systems. Because a path on one computer may not match a path on another one (because of different drive letters or drive names, or by being in different folders on different computers), most code that uses absolute paths is not portable.

Therefore, in this discussion, we will use relative paths. Instead of starting at the top of the file system on your computer, a *relative* path starts in the folder that contains your Python source file. We say that the path is *relative to the location of the source file*. That means any file we want to use or create resides either in the same folder as your Python source file or in a folder somewhere below that folder.

To see how this works, let's assume a folder structure like the one shown in [Figure 9-1](#).

Folder structure



Folder structure	Path to the text file
PythonSource.py	
MyDataFile.txt	'MyDataFile.txt'
MyFolder	
SomeDataFile.txt	'MyFolder/SomeDataFile.txt'
MySubFolder	
OtherDataFile.txt	'MyFolder/MySubFolder/OtherDataFile.txt'
MySubSubFolder	

Chapter 9 File input/Output

Figure 9-1. Example contents of a folder

In this example, we have an enclosing folder located anywhere on a computer. In that folder, there is a Python source file named PythonSource.py and a text file called MyDataFile.txt. In addition to these two files, there is also a folder named

called `MyDataFile.txt`. In addition to these two files, there is also a folder named `MyFolder`.

Within `MyFolder`, there is another data file called `SomeDataFile.txt` and a folder called `MySubFolder`. Within `MySubFolder`, there is a data file called `OtherDataFile.txt` and a subfolder. From the point of view of the `PythonSource.py` file, Figure 9-2 shows the relative paths to the three different data files.

Figure 9-2. Example contents of a folder with relative paths to text files 248

Chapter 9 File input/Output

In the simplest case, if you are running a Python program and you want to use a file in the same folder, then the path for the data file is simply the name of the file as a string.

In our example, if we were running `PythonSource.py` and we wanted to use the file `MyDataFile.txt`, we would specify the path as this string: `'MyDataFile.txt'`

However, if you want to use a file that is inside a folder where the Python program lives, then you specify the folder name, a *forward slash* (`/`)—which is more commonly referred to as simply a *slash*—and then the file name, all as a string. From `PythonSource.py`, we get to `SomeDataFile.txt` by using this path: `'MyFolder/SomeDataFile.txt'`

To go down two levels of folders and then find a file, specify the name of the first folder, followed by a slash, and then the next subfolder, followed by a slash, and then the name of the file. The following is the path to get to the `OtherDataFile.txt` file from `PythonSource.py`:
`'MyFolder/MySubFolder/OtherDataFile.txt'`

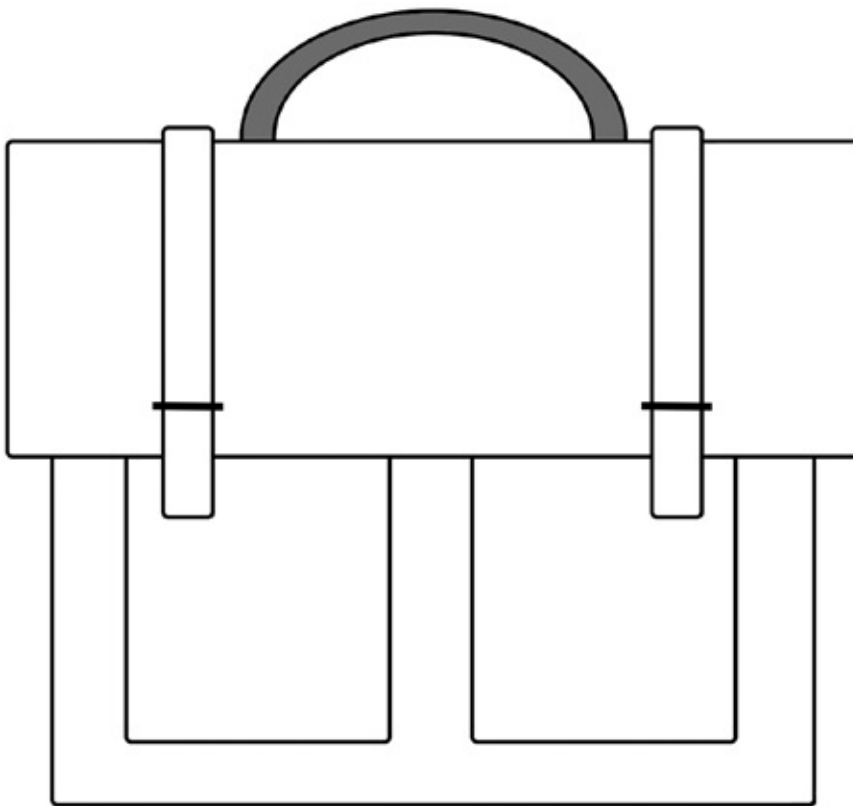
This can go on for any number of levels of subfolders. Just add a slash after every folder name, eventually placing the name of the file at the end. Through testing, we have found that although Windows uses the backward slash (`\`) as the folder separator character, using the slash (`/`) as the folder separator character in Python paths works correctly across operating systems. That means using the slash character in paths allows you to build platform-independent relative paths. That is, this approach allows you to build programs with relative paths on a Mac, and the program will be able to access files in the same relative folders on Windows (and vice versa) without changing code.

...mode (and vice versa), without changing code.

Reading from and Writing to a File

When you want to write to or read from a file, you first need tell the operating system that you want to *open* the file for reading or writing. To read, you read the contents of the file into a string variable. To write, you take the contents of a string variable and write that out to a file. When you are done reading or writing, you close the file.

249



Chapter 9 File input/Output

Reading text from a file requires three steps:

1. Open the file for reading
2. Read from the file (usually into a string variable)
3. Close the file

Writing text to a file requires three similar steps:

1. Open the file for writing
2. Write a string (usually from a string variable) to the file
3. Close the file

File Handle

Notice that whenever you deal with a file, you first need to open the file. In all operating systems, when a program opens a file, the operating system gives back a *file handle*.

Rather than give you a formal definition of what a file handle is, you should think of it like the illustration shown in Figure [9-3](#).

Figure 9-3. *Think of a file on the computer as a bag with a handle* 250

Chapter 9 File input/Output

The bag in Figure [9-3](#) represents a file. Any time you want to put something into the bag or take something out of it, you have to grab the handle. Similarly, any time you want to read from or write to a file on your computer, you have to use the file handle that the operating system gives you when you open the file. When you are done using the file, you have to close the file using the file handle; this is like releasing the handle of the bag.

The following is the core code needed to read from and write to a file. We will wind up *wrapping* this code into functions for you to use. For now, just pay attention to the basic steps involved.

Here is the code to read from a file:

```
fileHandle = open(filePath, 'r') # r for reading  
  
data = fileHandle.read() # read into a variable  
  
fileHandle.close()  
  
# text read in is now in the variable called data
```

And here is the code to write text to a file:

```
# text to be written is contained in the variable textToWrite  
  
fileHandle = open(filePath, 'w') # w for writing  
  
fileHandle.write(textToWrite) # write out text from a variable  
  
fileHandle.close()
```

Notice filePath (which is a string) and fileHandle (which is returned by the call to open). We use that file handle in the calls to read, write, and close.

Before attempting to read from a file, there is one more thing you need to do.

Obviously, you cannot read from a file that doesn't exist. So, you need to check that the file you want to read from actually exists before you attempt to read from it.

The Python os Package

In the same way that the random module provides a great deal of code for dealing with random numbers, there is a module that provides information about the operating system. It is called the os module. To use it, you first import it:

```
import os
```

251

Chapter 9 File input/Output

For now, we're only interested in one operation that can tell us whether a file exists.

Here's how to use it:

```
exists = os.path.exists(filePath) #returns a boolean
```

The call to os.path.exists returns True if the file exists, or False if the file does not exist.

OS MODULE

For anyone who is into uniX, or who wants to write the equivalent of shell scripts (for automation), the os module is extremely important. the os module allows you to do many uniX commands as python statements. here are just a few of the things the os module allows you to do:

- os.listdir: Generate a list containing the names of the entries in a

directory (folder)

- os.mkdir: Make a directory (folder)
- os.rename: rename a file
- os.walk: Generate the names of files in a directory (folder)
- os.getcwd: Get the current working directory (folder)
- os.chmod: Change the mode of a path

For a complete listing and detailed information, check the official python documentation at

<https://docs.python.org/2/library/os.html>.

Building Reusable File I/O Functions

We now have enough information to build three very useful, highly reusable functions.

We'll build the following:

- fileExists: Returns a Boolean to say whether a file with a given path exists or not
- writeFile: Takes a string of data and writes it to a file with a given path
- readFile: Reads the contents of a file and returns the contents to the caller

Chapter 9 File input/Output

Let's start by creating a new Python source file named `FileReadWrite.py`. We'll put the following code into it: `# FileReadWrite.py`

```
# Functions for checking if a file exists, read from a file, write to a file
```

```
import os
```

```
def fileExists(filePath):
```

```
    exists = os.path.exists(filePath)
```

```
    return exists
```

```
def writeFile(filePath, textToWrite):
```

```
    fileHandle = open(filePath, 'w')
```

```
    fileHandle.write(textToWrite)
```

```
    fileHandle.close()
```

```
def readFile(filePath):
```

```
    if not fileExists(filePath):
```

```
        print('The file, ' + filePath + ' does not exist - cannot read it.')
```

```
    return "
```

```
    fileHandle = open(filePath, 'r')
```

```
    data = fileHandle.read()
```

```
    fileHandle.close()
```

```
    return data
```

These functions provide very nice wrappers for the functionality. For example.

...these functions provides very nice wrappers for the functionality. For example,

now that we have written fileExists, we don't need to remember the details of the os module (that you need to use os.path.exists). Instead, we have built a simple function with a nice clean name of fileExists. We can reuse this function in any of our projects.

writeFile is very easy to use. You pass it a file path to write to and a string, and it writes the string to the file. If the file already exists, the older version of the file is completely overwritten by the new text.

The readFile function is also very straightforward. You pass in a path to a file, it checks to ensure that the file exists, and if so, does all the work to read all the text from the file and then returns that text to the caller. If the file does not exist, it prints an appropriate error message and returns the empty string to signify that there was no text to read.

253

Chapter 9 File input/Output

Example Using Our File I/O Functions

Let's work through an example of writing to and reading from a file. We'll start by selecting all the code we just built in FileReadWrite.py by using Command+A (Mac) or Control+A (Windows). Once all of it is selected, copy the code. Now open a new Python file. Paste the code into this new window. Save it with any name you want (be sure that the name ends in .py). Let's call this file TestFileIO.py.

Now we'll write some code to use these functions. The program we want to write will take a sample string, write it out to a file, and read it back in. Add the following after the three functions you pasted into this file: # Previous code from FileReadWrite pasted here

```
DATA_FILE_PATH = 'TestData.txt' # path to the file as a constant
```

```
stringToWriteOutToFile = 'abcdefghijkl' # contents could be anything,
```

```
this is just a test
```

```
writeFile(DATA_FILE_PATH, stringToWriteOutToFile)
```

```
writeFile(DATA_FILE_PATH, stringToWriteOutToFile)

stringReadInFromFile = readFile(DATA_FILE_PATH)

print('Read in: ', stringReadInFromFile)
```

When we save and run this program, we see this output in the Shell:

```
>>>
```

```
Read in: abcdefghijkl
```

```
>>>
```

What has happened here is that this code called our `writeFile` function to write out some text to a file. Then we used the `readFile` function to read from the same file back in, and saved the text in a different variable. Because we are using the same file path for reading and for writing, we specified the path to the file as a constant.

After running the program, if we look in the folder where this Python source file resides, we now see that a file named `TestData.txt` is present. Opening that text file in any text editor shows that the contents consist of the string we wrote out.

254

Chapter 9 File input/Output

Importing Our Own Modules

We could certainly use this approach of copying and pasting these three functions into any program that wants to perform any file I/O. But consider what happens if we find a bug in our `FileReadWrite.py` file, or if we want to add more functions to help read files in different ways. In either case, we would have to go back into every Python source file that incorporated these three functions and modify the code there to fix the bug and/or add functionality. There is a better way.

You have seen how to use the `import` statement to make a built-in Python package

available to your program. For example, you import the `random` package with this

available to our program. For example, you import the random package with this statement:

```
import random
```

When we import a package this way, we have to explicitly specify the name of the package when we make a call to a function in that package. For example, when we want to get a random number, we write this: `value = random.randrange(0, 10)`

That is very clear. It says that inside the random package, we want to call the `randrange` function.

In addition to being able to import built-in modules like the `random` and the `os` modules, we can use the `import` statement to import our own Python files. If we are building a program where we need to read from or write to a file, we can import our own `FileReadWrite.py` file. We can use the same `import` statement, like this: `import FileReadWrite`

Note an important thing to notice is that when you specify a `<moduleName>` to import, you do *not* specify the `.py` extension. I explain why it is done this way at the end of this chapter. For now, remember to remove the `.py` extension when specifying a file to import.

255

Chapter 9 File input/Output

After importing this way, you construct a line like this to write to a file:

```
FileReadWrite.writeFile('SomeFilePathToWriteTo', 'some test string')
```

However, there is another syntax available for the `import` statement. This alternative syntax allows you to specifically name which function(s) and/or variable(s) to import.

This is what it looks like:

```
from <moduleName> import <functionOrVariableName>
```

```
from <moduleName> import <functionOrVariableName>,  
<optionalFunctionOrVariableName>, ...
```

If you use the `from` syntax, then when you make a call to a function, you do *not* specify the package name, only the function name(s). The advantage is simplicity. For example, if you wanted to make a call to `readFile`, you would write this: `from FileReadWrite import readFile`

```
data = readFile('SomePathToAFileToRead')
```

The downside is twofold. First, if you import many source files this way, there is a chance of a name conflict. That is, it is possible that your main program file and one or more of your imported modules have a function or a variable that has the same name.

(In that case, whichever one was used last overrides the earlier one(s)). Second, if you import many modules, it may be confusing as to where a function name or a variable name came from, because it could come from one of many different files.

For the sizes of programs used in this book, neither of these should be considered a serious drawback. Most Python programmers use the `from` syntax when importing their own modules.

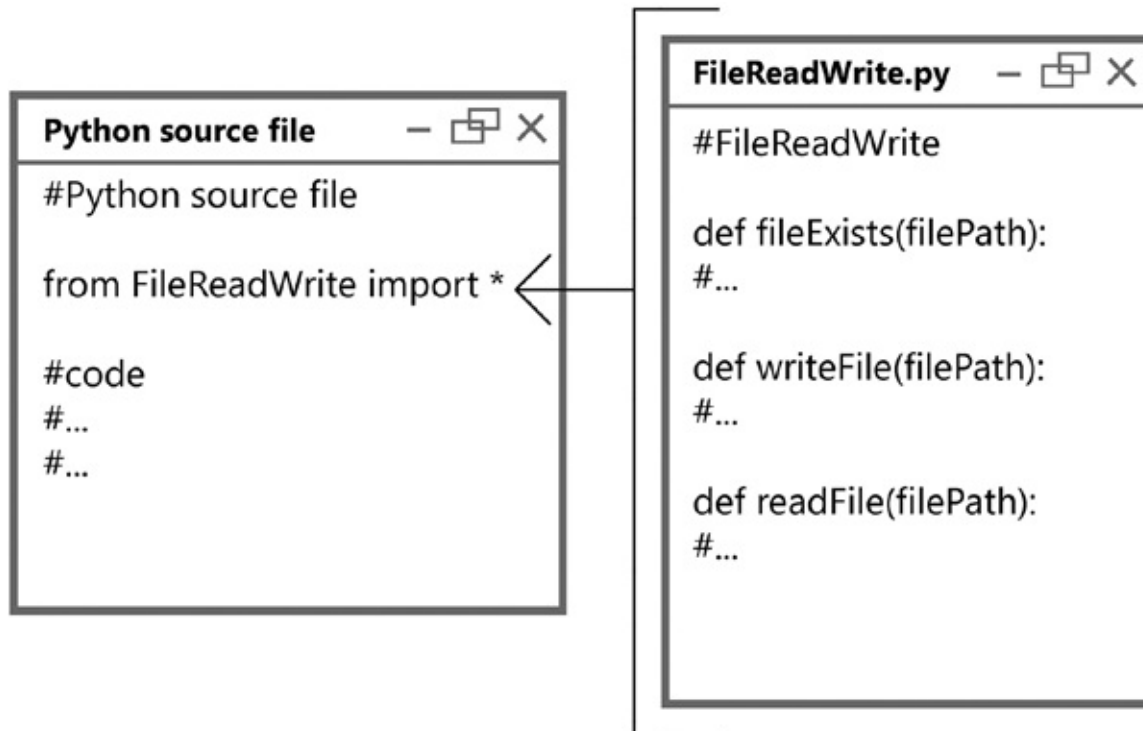
As an even simpler approach, you can tell Python to import an entire file of code using a line like this:

```
from <moduleName> import *
```

The asterisk (*) means bring in the entire contents of that file. For example, to import the `FileReadWrite` module and read a file, we would write this: `from FileReadWrite import *`

```
data = readFile('SomePathToAFileToRead')
```

Using this syntax essentially says to Python, “Bring in the full code of the `FileReadWrite` file as though I had written that code right here.” See [Figure 9-4](#).



Chapter 9 File input/Output

Figure 9-4. *Using the from statement to import contents of another file* If you plan to use most of or all of the code from an external Python file, use the asterisk to bring in the entire file. The case where you should name the functions individually is when the Python file you are importing is extremely large and you are only using a small number of functions.

For our purposes, using the asterisk syntax is fine and ensures that all functions in the external Python file are available to our code.

Building and using external Python files this way allows programmers to split up large programs into a number of files. Having the ability to import these types of Python source files allows you to build up files of reusable code and incorporate this code into multiple programs. Further, finding and fixing a bug in a file like this fixes the bug in every program that imports the file.

Saving Data to a File and Reading It Back

Now that we have built our three reusable functions in an external file, we can build a sample main program. Our goal is to write a program that counts the number of times it has run. To implement this, the program reads a data file that

contains the number of times the program has been run. Every time the program runs, it should read the file, add 1 to the number, and then rewrite the file.

257

Chapter 9 File input/Output

Immediately, we run into a problem: the first time we run the program, there is no file. So, we need to check for the existence of the file right from the outset. Let's first write the approach as pseudocode: If the file does not exist

Write a file with a 1 in it

Otherwise

Read the content of the file into a variable

Add 1 to the variable

Write out the value of the variable to a file

When we feel that our approach solves the problem, we turn the pseudocode into a real Python program. We'll take advantage of the three reusable functions we already built: `fileExists`, `readFile`, and `writeFile`. To make things clear, let's use a constant for the file path: # Increment test

```
from FileReadWrite import *
```

```
# Here is a constant - the name of the data file that we will use
```

```
throughout
```

```
DATA_FILE_PATH = 'CountData.txt'
```

```
# Main program - reads from file, increments a counter, writes to file
```

```
if not fileExists(DATA_FILE_PATH):
```

```
# The file was not found, this is the first time we are running the
```

```
program
```

program

```
print('First time - creating the data file.') # for testing

writeFile(DATA_FILE_PATH, '1')

else:

# The file was found. We have run this program before

count = readFile(DATA_FILE_PATH)

print('Found the file, data read was: ', count) # for testing

count = int(count)

count = count + 1

textToWrite = str(count)

writeFile(DATA_FILE_PATH, textToWrite)

print('This was run number:', count)
```

258

Chapter 9 File input/Output

Because we already have our reusable functions in `FileReadWrite.py`, we first use an import statement to bring in the code of the three functions that are found in that file.

Now let's walk through the logic of the program. The first time the program runs, the data file does not exist. So, we explicitly write out the string '1' to our external data file.

Every subsequent time we run the program, the file does exist, and the else clause will run. In the code of the else block, we read in the contents of the file (which is just a number, as text), we convert what we read into an integer and then increment it to add 1

to the number of times we have run the program. Finally, we convert the number

back to a string and write out the new value to the file.

Here is the output of several runs of the program:

```
>>>
```

First time - creating the data file.

This was run number: 1

```
>>> ===== RESTART
=====
```

```
>>>
```

Found the file, data read was: 1

This was run number: 2

```
>>> ===== RESTART
=====
```

```
>>>
```

Found the file, data read was: 2

This was run number: 3

```
>>> ===== RESTART
=====
```

```
>>>
```

Found the file, data read was: 3

This was run number: 4

```
>>> ===== RESTART
=====
```

```
>>>
```

Found the file, data read was: 4

This was run number: 5

As you can see from the output, the code following the if ran the first time that the program ran. In every subsequent run, the code following the else ran because the program found the data file.

259

Chapter 9 File input/Output

Building an Adding Game

Let's take these concepts and use them in a real program—a simple adding game for kids. Rather than build a large game in one shot, we'll split it up into four versions, adding complexity as we go.

Let's start by building the core part of the game, where we ask the user to add two integers and then see if the user answers correctly. We'll expand the program to allow for any number of questions and keep score. Then we'll expand it further to write out the score when we exit the program, and then read the score back in when we start up the program again. Finally, we'll modify the program yet again to write out and read back in several pieces of information about the game.

You'll write the first version of the game as a programming challenge.

Programming Challenge 1

Build a simple adding game. These are the details:

1. Allow the program to choose two random integers, each between 0 and 10.
2. Build and pose an addition question for the user, using the following form:

What is <num1> + <num2>?

3. Compare the user's answer to the correct answer.
4. Give feedback: correct or incorrect (if incorrect, show the correct answer).

Here is the output of two sample runs of the program:

```
>>>
```

```
What is: 10 + 3? 13
```

```
Yes, you got it!
```

```
>>>
```

```
>>>
```

```
What is: 10 + 5? 14
```

```
No, sorry, the correct answer was: 15
```

```
>>>
```

```
260
```

Chapter 9 File input/Output

Your code should look something like this:

```
# Adding game version 1
```

```
import random
```

```
firstNumber = random.randrange(0, 11)
```

```
secondNumber = random.randrange(0, 11)
```

```
correctAnswer = firstNumber + secondNumber
```

```
question = 'What is: ' + str(firstNumber) + ' + ' + str(secondNumber) + '? '
```

```
userAnswer = input(question)

userAnswer = int(userAnswer)

if userAnswer == correctAnswer:

    print('Yes, you got it!')

else:

    print('No, sorry, the correct answer was: ', correctAnswer)
```

The key to this program is generating two random numbers within the appropriate

range and adding them together so that you know what the correct answer should be.

Then you ask the user for their answer. Finally, you compare the correct answer to the user's answer and give appropriate feedback.

Programming Challenge 2

Once you have the first challenge code running correctly, the next steps are to modify it to allow the program to run in a loop and keep score. The details are as follows: 1. Add a score counter (start at zero).

2. Add a loop (to ask multiple questions).

3. If the user presses Return/Enter, exit loop.

4. If the user answers correctly, add 2 points to score.

5. Otherwise, for an incorrect answer, subtract 1 point from score.

6. Print the score.

7. When the user chooses to leave the program, say goodbye.

Chapter 9 File input/Output

Here is the output of a sample run of this version of the program:

What is: $7 + 6$? 13

Yes, you got it!

Your current score is: 2

What is: $1 + 4$? 5

Yes, you got it!

Your current score is: 4

What is: $4 + 1$? 5

Yes, you got it!

Your current score is: 6

What is: $10 + 1$? 11

Yes, you got it!

Your current score is: 8

What is: $3 + 8$? 9

No, sorry, the correct answer was: 11

Your current score is: 7

What is: $5 + 0$?

Thanks for playing

>>>

This is the solution to the challenge:

```
# Adding game version 2

import random

score = 0

# Main loop

while True:

    firstNumber = random.randrange(0, 11)

    secondNumber = random.randrange(0, 11)

    correctAnswer = firstNumber + secondNumber

    question = 'What is: ' + str(firstNumber) + ' + ' + str(secondNumber)

    + '? '
```

262

Chapter 9 File input/Output

```
userAnswer = input(question)

if userAnswer == "":

    break # user wants to quit

userAnswer = int(userAnswer)

if userAnswer == correctAnswer:

    print('Yes, you got it!')

    score = score + 2

else:

    print('No, sorry, the correct answer was: ', correctAnswer)
```

```
score = score - 1

print(Your current score is: ', score)

print()

print('Thanks for playing')
```

In this version, the changes are relatively small. We kept score (using a variable called score) by adding two points for each correct answer and subtracting one if an answer was incorrect. The important change was to put the main portion of the code in a while loop so that the user had as many addition questions as they wanted. We also checked for no answer (the empty string), which is the indication that the user wanted to quit the game.

Writing/Reading One Piece of Data to and from a File

In the next version of the game, let's add the ability to have persistent data by modifying the program so that when the user quits the program, the code writes out the score to a file. When the user chooses to start the program again, the score is read in from the file, and the program starts up using the previous score.

I'll present the code of this version, as follows, and then I'll explain the changes:

```
# Adding game version 3

# Save only the score

import random

from FileReadWrite import * # means import everything as though it were
typed here

263

Chapter 9 File input/Output

DATA_FILE_PATH = 'GameData.txt'

# Start up code
```

```
if not fileExists(DATA_FILE_PATH):

    score = 0

    print('Hi, and welcome to the adding game.')

else:

    score = readFile(DATA_FILE_PATH)

    score = int(score)

    print('Welcome back. Your saved score is:', score)

# Main loop

while True:

    firstNumber = random.randrange(0, 11)

    secondNumber = random.randrange(0, 11)

    correctAnswer = firstNumber + secondNumber

    question = 'What is: ' + str(firstNumber) + ' + ' + str(secondNumber)

    + '?'

    userAnswer = input(question)

    if userAnswer == ":

        break

    userAnswer = int(userAnswer)

    if userAnswer == correctAnswer:

        print('Yes, you got it!')

        score = score + 2
```



```
else:

    print('No, sorry, the correct answer was: ', correctAnswer)

    score = score - 1

    print('Your current score is: ', score)

    print()

    writeFile(DATA_FILE_PATH, str(score))

    print('Thanks for playing')
```

264

Chapter 9 File input/Output

In this version, the key changes are at the beginning and end of the program. The first thing we do in this program is bring in the code we developed earlier, which allows us to do file I/O. We import the code using this line: `from FileReadWrite import *`

That gives us access to the previously written `fileExists`, `readFile`, and `writeFile` functions. In order for this import to work correctly, the source file we are developing and the `FileReadWrite.py` file must be in the same folder.

Next, we define a constant for our file path. Any file name will work, so let's choose the very clear name of `GameData.txt`. Because the content is string data, we choose to use a `.txt` extension, meaning that it is filled with only text.

When the program starts, similar to our earlier `IncrementTest` program, we check for the existence (actually, the nonexistence) of our data file. If the data file does not exist, then we know this is the first time we are running the program. In that case, we welcome the user to our game and set our score variable to 0. If we find that the file does exist, we read the contents of the file into our score variable. Data read from a text file comes in as text in the same way that input produces text. Therefore, we have to convert the score to an integer. Then we welcome the user back to the game and tell them their previous score.

The central part of the code is identical. The user plays as many rounds as they like.

When the user is ready to quit the program, we take the current score and write it out to the data file. Finally, we thank the user for playing.

The data file is created (or updated) in the same folder as the source file and the FileReadWrite.py file. You can easily open it and view the contents with any text editor.

When we look at the contents of the file after playing any number of rounds, all we see is a text version of the most recent score. If you write to a file that already exists (any run after the first run), the previous contents of the file are overwritten.

265

Chapter 9 File input/Output

Writing/Reading Multiple Pieces of Data

to and from a File

In the final version of the game, we'll want to keep track of, write out, and read back four pieces of information:

- User name

- Score
- Number of problems tried
- Number of problems answered correctly

To write out and read back multiple pieces of information, we need two more built-in functions: split and join.

The join Function

Let's start with the join function. The data we want to write to a file must be one long text string. If we want to write out multiple pieces of data, we need to build a string that incorporates all of them. We'll do this in two steps: 1. Take all data that we want to save (converting any numbers to

string versions) and then create a list containing the data.

2. Combine the list into a string.

The purpose of the join function is to take a list (of strings) and concatenate all elements to create a single long string. In the resulting string, each piece of original data is separated by a character of your choice. The comma is the most typical character used to separate this type of data. join is a string operation, but it has an odd syntax. It is most often used in an assignment statement, like this:
<string> = <separatorCharacter>.join(<list>)

join takes the list (of strings) and creates a new string by concatenating all the elements of the list, separated by the given separator character. Here is an example: >>> myList = ['abc', 'de', '123', 'fghi', '-3.21']

```
>>>
```

```
>>> # Use a comma as a separator character
```

```
266
```

Chapter 9 File input/Output

```
>>> myString = ','.join(myList)
```

```
>>>
```

```
>>> print(myString)
```

```
abc,de,123,fghi,-3.21
```

```
>>>
```

Here you can see that the join function has taken a list of string data and created a single comma-separated string.

The split Function

The other built-in function is the split function, which takes a string and splits it at every point where it finds a given separator character, into multiple pieces of

data in a list. `split` is typically used in an assignment statement, like this: `<list> = <string>.split(<separatorChar>)`

Here is an example:

```
>>>
```

```
>>> myString = 'abc,de,123,fg,hi,-3.21'
```

```
>>>
```

```
>>> myList = myString.split(',')
```

```
>>>
```

```
>>> print(myList)
```

```
['abc', 'de', '123', 'fg,hi', '-3.21']
```

```
>>>
```

Because `split` is an operation on a string, we can use it after we read in data from a file and separate out the individual pieces of data that were used to make up the string when the file was written.

Because of the syntax, `split` and `join` are both considered string operations. `join` operates on a separator character, whereas `split` operates on a string to be broken apart.

But these operations perform complimentary or opposite actions. Think of it like this: `join` is passed a list and produces a string, but `split` takes a string and produces a list.

`join` is often used for writing out to a file, whereas `split` is often used for reading data in from a file.

267

Chapter 9 File input/Output

Final Version of the Adding Game

Now, in addition to remembering the score, let's modify the game further by keeping track of three more pieces of data. The first time we play the game, we'll ask for and remember the user's name. We'll also remember the number of problems the user has seen and the number of problems the user has answered correctly.

To keep track of this additional information, we'll add three more variables:

userName, nProblems, and nCorrect. When the user chooses to quit the program (by pressing Return or Enter), we'll add some code to write the information we want to remember out to a file. As a format for the content of the file, we'll use the following: <name>,<score>,<nProblems>,<nCorrect>

For example, after playing the game once and answering 14 out of 15 questions correctly, the file for our user Joe Schmoe looks like this:

```
Joe Schmoe,27,15,14
```

Here is the code of the final version that implements these changes. The modifications are significant, but everything should be understandable:

```
# Adding Game version 4
```

```
# Saving lots of data
```

```
import random
```

```
from FileReadWrite import * # means import everything as though it were  
typed here
```

```
DATA_FILE_PATH = 'AddingGameData.txt'
```

```
# Main program starts here
```

```
if not fileExists(DATA_FILE_PATH):
```

```
    userName = input('You must be new here, please enter your name: ')
```

```
    ^
```

```
score = 0  
  
nProblems = 0  
  
nCorrect = 0  
  
print('To quit the game, press RETURN/ENTER and your info will be saved')  
  
print('OK', userName, "let's get started ...")  
  
print()
```

268

Chapter 9 File input/Output

else:

```
savedDataString = readFile(DATA_FILE_PATH) #read the whole file into a  
variable
```

```
savedDataList = savedDataString.split(',') # turn that into a list
```

```
userName = savedDataList[0]
```

```
score = savedDataList[1]
```

```
score = int(score)
```

```
nProblems = savedDataList[2]
```

```
nProblems = int(nProblems)
```

```
nCorrect = int(savedDataList[3]) # can do both in a combined step
```

```
print('Welcome back', userName, 'nice to see you again! ')
```

```
print('Your current score is: ', score)
```

```
print()
```

```

# Main loop

while True:

    firstNumber = random.randrange(0, 11)

    secondNumber = random.randrange(0, 11)

    correctAnswer = firstNumber + secondNumber

    question = 'What is: ' + str(firstNumber) + ' + ' + str(secondNumber)
    + '? '

    userAnswer = input(question)

    if userAnswer == " ":
        break

    userAnswer = int(userAnswer)

    nProblems = nProblems + 1

    if userAnswer == correctAnswer:

        print('Yes, you got it!')

        score = score + 2

        nCorrect = nCorrect + 1

269

Chapter 9 File input/Output

else:

    print('No, sorry, the correct answer was: ', correctAnswer)

    score = score - 1

```

```
print('Your current score is: ', score)

print()

print('Thanks for playing')

print()

print('You have tried', nProblems, 'problems and you have correctly
answered', nCorrect)

# Make a list of the useruserName, userScore, nProblems, nCorrect then

# create a string from that using join

dataList = [userName, str(score), str(nProblems), str(nCorrect)]

outputText = ','.join(dataList)

writeFile(DATA_FILE_PATH, outputText)
```

We chose to use a different file path from the previous version because this version writes out different data.

As with the previous version, we start by checking to see if our data file exists. If it does not, we conclude that this is the first time the user is playing the game. If the file does exist, we assume the user has played the game before and we need to read in the data from the file.

If this is the first time playing the game, we give a greeting to the user, ask their name, and initialize the variables score, nProblems, and nCorrect all to zero.

If the user has played the game before, we read the contents of the file using the readFile function we developed earlier and use the split function on the data that we read in. That generates a list. In the list that is created, we know that element 0

contains the user's name, element 1 contains the score, element 2 contains the number of problems, and element 3 contains the number of problems answered correctly. We extract these pieces of information from the list and store them

into the same three variables. Finally, we print out some messages to welcome the user back and tell them their current score.

The central part of the game is nearly identical, except we have added code to increment the number of problems asked and to increment the number of problems answered correctly.

270

Chapter 9 File input/Output

When the user chooses to quit the program, we tell them the number of problems they have seen and the number they answered correctly. Then we take the data we want to save, build a list out of it (while ensuring that each piece of information is converted to a string), and then use the join function to turn that list into one comma-separated string. Finally, we write that string out to the data file using the writeFile function we developed earlier.

Though this program was set up to write out and read in four pieces of data, you can use these same techniques to write out, and later read back in, any number of pieces of data.

Writing and Reading a Line at a Time with a File

In the code we have developed so far, you have seen how to write a text file from a single variable or read a text file into a single variable. But there are times where we want to write data to a file a line at a time or read data from a file a line at a time. Here are five more small functions that should be added to the bottom of the earlier file, FileReadWrite.py. These additional functions allow us to write and read files this way: # (Earlier code for fileExists, writeFile, readFile)

#

Functions for opening a file, writing & reading a line at a time, and closing the file

```

def openFileForWriting(filePath):
    fileHandle = open(filePath, 'w')
    return fileHandle

def writeALine(fileHandle, lineToWrite):
    # Add a newline character '\n' at the end and write the line
    lineToWrite = lineToWrite + '\n'
    fileHandle.write(lineToWrite)

def openFileForReading(filePath):
    if not fileExists(filePath):
        print('The file, ' + filePath + ' does not exist - cannot read it.')
    return “
271

```

Chapter 9 File input/Output

```

fileHandle = open(filePath, 'r')
return fileHandle

def readALine(fileHandle):
    theLine = fileHandle.readline()

    # This is a special check for attempting to read past the end of the
    file (EOF).

    # If this occurs, let's return something unusual: False (which is not a
    string)

```

```

# If the caller wishes to check, their code can easily detect the end
of the file this way

if not theLine:

    return False

# If the line ends with a newline character '\n', then strip that off
the end

if theLine.endswith('\n'):

    theLine = theLine.rstrip('\n')

    return theLine

def closeFile(fileHandle):

    fileHandle.close()

```

Here is the basic idea of how to use these functions. If you have a case where you want to write data one line at a time, you have to follow the same three steps outlined earlier: open the file, write to the file, and close the file. Rather than doing the three steps in a single call (as we did with `writeFile`), here we use three separate functions to implement the three steps:

- `openFileForWriting`: Opens the file for writing.

- `writeALine`: Call this as many times as you want; each call writes a line of text.

- `closeFile`: Closes the file.

Feel free to read the code of these three functions, but once you know that they work correctly, you do not need to remember the details of the implementation. By adding these functions to the earlier `FileReadWrite.py` file, these functions become part of your 272

Chapter 9 File input/Output

reusable library. As discussed earlier in the chapter, to make these functions available in your Python source file, you import the FileReadWrite package using this line: `from FileReadWrite import *`

Let's look at an example of how you might use these new functions. In the following code, we will write three lines of text to a file named `MultiLineData.txt`.

The `FileReadWrite.py` file must be in the same folder as our source file because it is imported into the source file: `# Write multiple lines of text to a file`

```
from FileReadWrite import *
```

```
DATA_FILE_PATH = 'MultiLineData.txt'
```

```
myFileHandle = openFileForWriting(DATA_FILE_PATH)
```

```
data1 = 'Here is some data as a string'
```

```
writeALine(myFileHandle, data1)
```

```
data2 = 'Here is a second line of string data'
```

```
writeALine(myFileHandle, data2)
```

```
# Could have some code join several pieces of data into a single string
```

```
data3 = '123,Joe Schmoe,123.45,0'
```

```
writeALine(myFileHandle, data3)
```

```
closeFile(myFileHandle)
```

The code should be very clear. We open the file for writing, write three lines of text, and then close the file. The key to using these functions is that the call to open the file returns a file handle. You then use this file handle in every call to `writeALine`. When you are done writing, you close the file with a call to `closeFile` passing in the file handle.

Running this program creates a text file (in the same folder as the program) called MultiLineData.txt, with the following contents: Here is some data as a string

Here is a second line of string data

123,Joe Schmoe,123.45,0

273

Chapter 9 File input/Output

If we have a file such as this MultiLineData.txt and we want to read it into our program, we perform very similar steps: open the file for reading, read the data, close the file. We use calls to the following functions to read in the data: •

openFileForReading: Opens the file for reading.

- readALine: Call this as many times as you need to; each call reads in a line of text.

- closeFile: Closes the file.

To read in the data that was previously written out to our MultiLineData.txt file, we could have a program like this: # Read in multiple lines of text

```
DATA_FILE_PATH = 'MultiLineData.txt'
```

```
myFileHandle = openFile(DATA_FILE_PATH)
```

```
data1 = readALine(myFileHandle)
```

```
print(data1)
```

```
data2 = readALine(myFileHandle)
```

```
print(data2)
```

```
data3 = readALine(myFileHandle)
```

```
print(data3)
```

Could add code to split data3 into several different pieces of data

closeFile(myFileHandle)

Again, the key concept is the file handle that is generated by a call to

openFileForReading. We use this file handle in every call to readALine. When we are finished reading, we call closeFile, passing in the file handle.

Having all these functions bundled into a separate file (FileReadWrite.py) makes for a nice reusable package. We only need to know the names of the functions and what data each one needs to be passed.

274

Chapter 9 File input/Output

Example: Multiple Choice Test

Let's put many of these concepts together and build a useful example program. We'll create a program that allows the user to take a multiple-choice test. The interesting thing about the program is that we'll write it in a way that it can be used to pose any number of questions on any topic.

Definition *Content independence* is a program's ability to use data that is not built into the program.

We will build our multiple-choice test program in a content-independent way by

having the questions and answers in an external text file. If we define and use a clear layout for this text file, the program can be used as a generic "engine" that runs through any number of questions on any topic.

We'll first make the decision that each multiple-choice question has four possible answers. We'll define a layout for our questions file, like this: <Test title line>

<Number of questions>

<Question 1>

<Correct answer for question 1>
<Incorrect answer 1 for question 1>
<incorrect answer 2 for question 1>
<Incorrect answer 3 for question 1>
<Question 2>
<Correct answer for question 2>
<Incorrect answer 1 for question 2>
<incorrect answer 2 for question 2>
<Incorrect answer 3 for question 2>
...
<Question n>
<Correct answer for question n>
<Incorrect answer 1 for question n>
<incorrect answer 2 for question n>
<Incorrect answer 3 for question n>

275

Chapter 9 File input/Output

In this layout, the first line of the file is a title line that will be presented to the user.

The second line contains a text version of an integer that will tell us how many questions there are in the test.

After that, each question is made up of a grouping of five lines. The first line of each group is the question itself. After that is the correct answer to the question.

Each group is the question itself. First that is the correct answer to the question. Then there are three incorrect or “distracter” answers.

Here is a sample test file with four questions:

Stupid answers quiz

4

What color was Washington's white horse?

White

Blue

Red

Beige

How many green Chinese pots are there in a dozen?

12

1

10

-6

What is the state song of Alabama?

Alabama

New Jersey is the place for me

My home is in Australia

I like monkeys

What is the first verb in the Pledge of Allegiance?

pledge

allegiance

snorkel

For each question, the program reads in five lines, poses the question, randomizes the answers, and presents the randomized answers. It waits for a user response, checks to see whether the user got the question correct or not, and gives appropriate correct or incorrect feedback.

276

Chapter 9 File input/Output

During the test, the program keeps and presents a running score. At the end of the test, the program calculates the percentage correct.

Here is the code for the multiple-choice test program:

```
# Multiple choice test

import random

from FileReadWrite import *

FILE_PATH = 'MultipleChoiceQuestions.txt'

LETTERS_LIST = ['a', 'b', 'c', 'd']

# Open the file for reading, read in the title line
fileHandle = openFileForReading(FILE_PATH)

titleText = readALine(fileHandle)

# Find out how many questions there will be
nQuestions = readALine(fileHandle)

nQuestions = int(nQuestions)

nprint('Welcome! This test is:')
```

```

print( welcome. This test is. )

print()

print(titleText) # print whatever title we got from the file

print()

print('There will be', nQuestions, 'questions.')

print()

print("Let's go ...")

print()

score = 0

# Each time through the loop, handle a single question
for questionNumber in range(0, nQuestions):

    questionText = readALine(fileHandle) # read a line of a question

    answers = []

    for i in range(0, 4):

        thisAnswer = readALine(fileHandle) # read each answer

        answers.append(thisAnswer)

277

```

Chapter 9 File input/Output

```

correctAnswer = answers[0] # save away the correct answer

random.shuffle(answers) # randomize the 4 answers

indexOfCorrectAnswer = answers.index(correctAnswer) # see where the
correct answer is

```

correct answer is

present the question and the four randomized answers

print

print(str(questionNumber + 1) + '. ' + questionText) #ask question

for index in range(0, 4):

thisLetter = LETTERS_LIST [index]

thisAnswer = answers[index]

thisAnswerLine = “ + thisLetter + ') ' + thisAnswer

print(thisAnswerLine)

print

Ensure that the user enters a valid letter answer

while True:

userAnswer = input('Your answer (a, b, c, or d): ')

userAnswer = userAnswer.lower() # convert usersAnswer to lowercase

if userAnswer in LETTERS_LIST: # valid answer

break

else: # invalid answer

print('Please enter a, b, c, or d')

Find the index associated with the user's answer

The following maps a to 0, b to 1, c to 2, d to 3

indexOfUsersAnswer = LETTERS_LIST.index(userAnswer)

“ a b c d ”

```

# Give feedback

if indexOfCorrectAnswer == indexOfWorkersAnswer:

    score = score + 1

    print('Correct!')
else:

    print("Sorry, that's not it.")

    correctLetter = LETTERS_LIST[indexOfCorrectAnswer]

    print('The correct answer was: ', correctLetter + ') ' +
correctAnswer)

```

278

Chapter 9 File input/Output

```

print()

print('Your score is:', score)

# Done, show the percent correct and close the file

pctCorrect = (score * 100.) / nQuestions

print()

print('All done! You got:', str(pctCorrect) + '% correct')

closeFile(fileHandle)

```

I won't go into all the details of this program because it is well commented. The only tricky part is finding the index of the correct answer and matching it up to the index of the answer the user chose. We use the built-in list index operation to find the index of where the correct answer wound up in our randomized list. We also use the index operation to map the user's letter answer (*a*, *b*, *c*, or *d*) into an

index (0, 1, 2, 3). We compare the user's choice index to the correct index to see if the user answered the question correctly. When we run the program, the output looks like this: >>>

Welcome! This test is:

Stupid answers quiz

There will be 4 questions.

Let's go ...

1. What color was Washington's white horse?

- a) Red
- b) White
- c) Blue
- d) Beige

Your answer (a, b, c, or d): b

Correct!

Your score is: 1

279

Chapter 9 File input/Output

2. How many green Chinese pots are there in a dozen?

- a) 10
- b) -6
- c) 1
- d) 12

Your answer (a, b, c, or d): d

Correct!

Your score is: 2

3. What is the state song of Alabama?

a) New Jersey is the place for me

b) I like monkeys

c) Alabama

d) My home is in Australia

Your answer (a, b, c, or d): c

Correct!

Your score is: 3

4. What is the first verb in the Pledge of Allegiance?

a) snorkel

b) allegiance

c) pledge

d) I

Your answer (a, b, c, or d): a

Sorry, that's not it.

The correct answer was: c) pledge

Your score is: 3

All done! You got: 75.0% correct

>>>

The ability to put data (such as the question data for this program) in an external file is a very powerful technique. Content-independent programs like this can have very wide applicability.

280

Chapter 9 File input/Output

A Compiled Version of a Module

If you have run a program that imports the `FileReadWrite` module that we built in this chapter, you might notice that in the same folder, there is now a folder named `__pycache__`. In that folder is a file named `FileReadWrite.cpython-3x.pyc`. I'll explain what this is.

Earlier I said that when you run a Python program, the Python compiler reads your code and “compiles” it into a simpler form called *bytecode*. The bytecode version is what actually runs on the computer. Whenever a program imports another Python file, the imported file must also be compiled. To simplify, let's say that we have a program called `A.py` that imports `B` (from the Python source file `B.py`). In this case, both `A.py` and `B.py` must be compiled. Python does this in a very smart way. Because you are typically editing `A.py` before your run, it makes sense to recompile that file every time you run.

But most of the time, `B.py` does not change. Therefore, when Python sees a statement to import `B`, it checks to see if `B.pyc` exists in the folder named `__pycache__`. If that file does not exist, it compiles `B.py` and produces a compiled bytecode version named `B.pyc`.

The `.pyc` extension stands for *Python compiled*. The next time you go to run your `A.py` program, Python sees the `B.pyc` file and uses that version of `B` since it has already been compiled. This results in faster compile times (the time between when you say, “Run,”

and when the program actually starts to run).

If you make a change to `B.py`, Python must recompile that file and produce a new

B.pyc. The way it knows when to do this is simple and clever. Whenever Python finds an import statement asking to import a module, it checks to see if there is a related .pyc file. If there is, it compares the last edited date/time of the .py file against the last edited date/time of the .pyc file. If the date/time of the .py file is after the date/time of the related .pyc file, it knows that the source file has been changed and it must recompile the .py file to produce a new .pyc file.

When you write a program that imports FileReadWrite.py, Python looks for

FileReadWrite.pyc. in the __pycache__ folder. If that file does not exist in that folder, Python reads your FileReadWrite.py, compiles it, and produces the bytecode version of the file: FileReadWrite.pyc. It now uses the bytecode version of the file.

Now it should be clearer that when you write an import statement, you only specify a module name (such as FileReadWrite) and leave off the file extension.

281

Chapter 9 File input/Output

Summary

In this chapter, you learned how to write to and read from a file. To use a file, you must first identify the file that you want to use by specifying its path as a string. Reading from or writing to a file involves three steps: open the file, read from or write to it, and close the file. Whenever you programmatically open a file, the operating system gives you back a file handle that you use in subsequent calls to write or read data. When you are finished, you must close the file, again using the file handle. You saw that Python's built-in os package contains many useful operating systems functions.

We then built a set of three reusable functions: fileExists, writeFile, and

readFile. Given these functions, we built a small example that used those functions to write a string of text to a file and read it back in. To make the functions truly reusable, we learned how to keep the functions in a separate Python source file and use the import statement to bring an external file into our code.

We then built four versions of a simple children's adding program. The final

we then built four versions of a simple children's adding program. The final version was able to save its state by writing out and reading back multiple pieces of data used by the program. This allowed the program to pick up right where the user left off. Internally, we used two new functions: join (to combine the data into a single string before writing to a file) and split (to read back the data from the file and break it up into the original data).

Our final topic on file I/O was the ability to read and write a line of data at a time with a file. Although we still must use the same three steps of opening a file, reading or writing it, and closing the file, we built a set of functions for these three steps. You saw how to use the file handle provided when you open the file in subsequent calls to read a single line of data or write a single line of data and then to close the file. This technique allows us to read and write large quantities of data using text files. I provided an example of building a generic multiple-choice testing program that is completely content independent by moving the data into a text file.

Lastly, we discussed how Python creates a compiled version of a Python module that is imported into other Python source files.

282

CHAPTER 10

Internet Data

In the previous two chapters, we discussed different ways to get and manipulate strings.

That makes this our third chapter on strings. Earlier, I talked about how a program can get text input from the user by using a call to input. In the previous chapter, I showed how a program could get data from and save data to a file. But there is another place where programs can get text data from: the Internet!

This chapter discusses the following topics:

- Request/response model
- Getting a stock price
- Pretending to be a browser

- API
- Requests with values
- API Key
- Example program to get stock price information using an API
- Example program to get weather information
- URL encoding

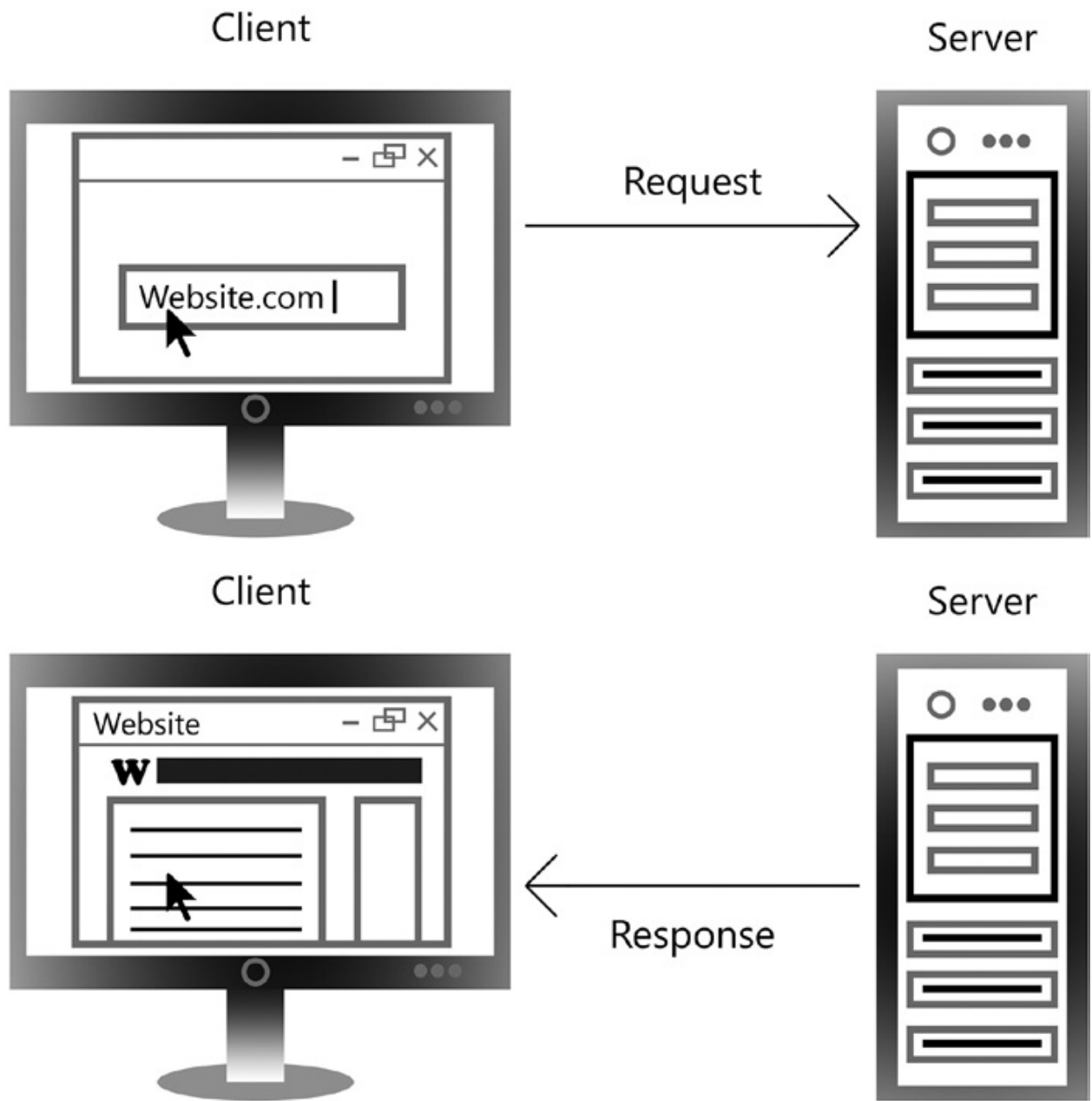
Request/Response Model

When you use a browser to go to a web site, you enter a URL (which stands for *Universal Resource Locator*), you press Enter or Return, and soon you see a nicely formatted web page. I'll explain what happens behind the scenes when you use a browser this way (see Figure [10-1](#)).

283

© Irv Kalb 2018

I. Kalb, *Learn to Program with Python 3*, https://doi.org/10.1007/978-1-4842-3879-0_10



Chapter 10 Internet Data

Figure 10-1. Client computer making a request to a server

When you use any browser on any device (computer, tablet, phone, and so on) to go to a web site, the system you are using is called the *client* and the computer that hosts the web site is called the *server*. After typing a URL, when you press Enter or Return the browser on the client makes a query called a *request* that is sent across the Internet.

Assuming that the URL is well formatted, the request is sent to the appropriate server of the site given in the URL. The browser running on the client then waits for an answer to the query from the server.

Having received the request, the server does whatever it needs to do to answer the request. In the case of a typical request to display a web page, it prepares and formats its answer. The answer is known as a *response* (see [Figure 10-2](#)).

Figure 10-2. Server sending back a response to a client computer 284

Chapter 10 Internet Data

The response is made up of text, formatted in a language called HTML (for *HyperText Markup Language*). Additionally, the response can contain other data, such as pictures, sounds, videos, and so forth. For this discussion, we'll concentrate only on the HTML

portion of the response. When the response is ready, it is sent back from the server to the client's browser.

The browser reads through the returned HTML, formats the resulting page, and shows the page in the browser window.

This sequence is commonly known as the *request/response model*. The key thing to realize here is that the browser sends a string (the request URL) to the server and then receives a string (the response HTML) from the server.

If you think about what the browser is doing, the steps are very similar to the

way we make a function call. The main difference is that instead of calling a function in its own program, or a built-in function in its own language, the action of the browser making a request is like making a function call, but across the Internet. Similar to what happens when you make a call to a function in Python where the caller waits until the function is finished, the browser waits until the server returns a response.

Note In this chapter we will discuss a number of Internet web sites. as of this writing (summer 2018), all the examples and related sample code work perfectly.

But some of these examples may fail later if the companies that provide them

change the way things are done on their sites. the important thing to learn is the generalized underlying concepts.

Getting a Stock Price

Suppose you want to find out the current stock price of a company. First, you have to know the stock symbol. (If you don't know a company's stock symbol, you can find it by using a search engine and entering "stock symbol for xxx," where xxx is the name of the company.) Let's say you wanted to find the price of Apple stock, whose symbol is AAPL. There are a number of sites that you can go to in order to get the price of a stock.

One very useful one is the site of the NASDAQ stock exchange. You can use any browser and enter the following URL to get the current price of Apple stock: <https://www.nasdaq.com/symbol/aapl>

285

Chapter 10 Internet Data

When you press Enter or Return, the browser sends the request. The NASDAQ server receives the request and looks up current information about Apple stock. It then builds the appropriate HTML code to send back to the browser. The browser receives all of the returned HTML as a single string, formats the information, and displays the page on the screen. You see the current stock price for Apple near the top of the page. You also see the change and percent change of the stock price, some key stock data (such as the best bid/ask price, 1-year

target, today's high value, and many more), along with an interactive chart of previous prices, and even some recent news about Apple.

The underlying HTML code that makes up any web page is available to view in the

browser. In Google Chrome, you can see it by right-clicking any web page and, from the context menu, choosing View Source. In Safari, you can see it by clicking Develop ►

Show Page Source.

In the source of the page, you can see that a huge amount of text has been returned from this request—more than a thousand lines of text. Somewhere in there is the price of Apple stock.

Let's remember what we just did and relate it back to the world of Python.

Pretending to Be a Browser

In Python, we can write a program that effectively pretends to be a browser. That is, instead of the browser making the request to get information from the NASDAQ site (or other similar site), we can write a Python program that can make the same request.

The response—all the HTML text—will come back to our program, and we can save all that text into a variable. But instead of painting the entire page like the browser does, we'll just look for the specific information we want to find: the price of a stock.

Python has a module called `urllib` and a submodule called `request` that provide

the code needed to allow programs to make requests over the Internet. As with other packages we have discussed, you bring this package into your code with this line: `import urllib.request` Once you have done that, the following two lines can be used to make the request across the Internet:

```
connection = urllib.request.urlopen(<URL>)
```

```
responseString = connection.read().decode()
```

```
...
```

Chapter 10 Internet Data

First, we call `urllib.request.urlopen`, specifying the URL you want to connect to.

This call returns a value similar to the file handle we used when dealing with file I/O in [Chapter 9](#). Across the Internet, this call returns what you can think of as a connection.

Assuming that works, the returned connection value is then used in a call to the `read` function to get the data from the given URL. However, we need to do one more step.

The data that is read from a site is often *encoded* (I'll explain more about this later in this chapter). We have to *decode* the data to turn it into a string that we can use in a Python program. The resulting string is stored into a variable.

For example, to get stock-quote information for Apple, we can use this code:

```
nasdaqAppleURL = 'https://www.nasdaq.com/symbol/aapl'

connection = urllib.request.urlopen(nasdaqAppleURL)

responseString = connection.read().decode()
```

When these lines run, the variable `responseString` is set to the exact same

underlying HTML information that the browser gets for that page when it makes the request. We know that the stock price is included somewhere in the returned text because when the browser makes the request and gets the response, we see the stock price on the screen. If we were to spend a lot of time analyzing the HTML we got back as a response, we could find an identifying tag that precedes the actual price string. In this case, there is an HTML tag `<div id="qwidget_lastsale" class="qwidget-dollar">` right before the price of the stock. Knowing that this exists in the HTML, we can write some code that reaches into the long HTML string and using a slice, grabs just the characters that make up the price.

This technique is called *screen scraping*. That is, we are taking data that is

intended for drawing on the screen of the computer and reading through it to find the particular piece or pieces of information we want.

This approach does work, but it can fail. The problem is that the code to find our specific piece of information is based on knowledge of how the page looks on the screen today. The company that owns the base URL (NASDAQ, in this example) could decide at any time to change the way the page is laid out. Further, we were able to find the piece of information because we noticed that the specific tag was built into the HTML of the page. At any point, the company responding to a request could also change the internal tag—for example, to `<div id="qwidget_lastprice" class="qwidget-dollar">`. In either of these cases, the program would break because the page is no longer formatted the way the program expects.

287

Chapter 10 Internet Data

The bottom line is that although this technique works (temporarily), it is *not* a good way to get information from the Internet into your program. Let's look at a much better way.

API

Companies that want to share their data make it available to computer programmers by publishing a set of guidelines for retrieving this type of data in a much more efficient way. These guidelines are called an API.

Definition an *apI* (short for *Application Programming Interface*) is a set of URLs and parameters that is designed to be called by programs across the Internet.

The idea is that instead of receiving a full page of HTML designed for display, using an API programmers can ask for and get just the information they want and use it in their programs. For example, there is a company named Alpha Vantage that has a set of APIs for retrieving data about stocks, physical and crypto currencies, stock indicators, and sector performances. It is fully documented at www.alphavantage.co/documentation/.

To get a stock quote, there is a single URL with a number of different parameters you can specify. The base URL is www.alphavantage.com/query.

But in order to make a useful request, you must also specify a number of additional values.

Requests with Values

If we think of the request/response model being like a function call, it would make the model more complete if we could pass data with a request, just as we pass values when we make a call to a Python function. It turns out that you can do exactly that. However, passing data with a URL has a different syntax than the way we do it in Python. The reason for this difference is that requests over the Internet are independent of the programming language. When passing data with a URL, there has to be a very general syntax. That syntax looks like this: `http://<URL>?<parameterName1>=<value1>&<parameterName2>=<value2>` etc.

288

Chapter 10 Internet Data

At the end of the base URL, you add a question mark to indicate that more information is coming. Following the question mark, you build any number of sequences of the form `parameterName=value` (with no spaces). Each grouping is commonly known as a *name/*

value pair. After the first name/value pair, additional `parameterName=value` pairs must be separated by an ampersand character (&). This syntax is different from how we make a call in Python, where we pass an argument simply as a variable or a value. With a URL request, you must supply the exact name of each parameter that the site expects. The names of these parameters are typically given in the documentation of the API. Each value to be passed must be a string, but without quotes, because the entire URL is specified as one long string. Some example name/value pairs in a URL could look like this: ?
`firstname=Joe&lastname=Schmoe&age=36`

These name/value pairs say that for the `firstname` parameter, use a value of Joe; for the `lastname` parameter, use a value of Schmoe; and for the `age` parameter, use a value of 36.

Now we are almost ready to build the full URL to get a stock quote using the Alpha Vantage API. According to its documentation, to get a stock quote we

need to specify three pieces of information. The first two are very straightforward.

The first piece of information tells the site what “function” to perform. In this case, we want a stock quote (there are many other options). To indicate that, we need to add this to the URL: `?function=BATCH_STOCK_QUOTES`

Next, we need to specify which stock symbol to get a quote for. To do that, we add the stock symbol for Apple like this: `&symbols=AAPL`

The last piece of information is an identifier called an API key.

API Key

Companies (and government agencies) that provide data via APIs often provide the data free to programmers who use their extensive APIs. But in order to prevent overuse and/

or potential malicious intent, organizations often require that you obtain an identifying key from them. An *API key* is a unique string that identifies you as the person whose code is making a request. API keys are typically given out for free by filling out an online ²⁸⁹

Claim your API Key

Claim your free API key with lifetime access. We do not send promotional or marketing materials to our users - we will reach out only in the event of launching new API features or server-side updates.

First Name:

Last Name:

Which of the following best describes you?

Email:

GET FREE API KEY

Chapter 10 Internet Data

form. Once you get an API key, you need use it in all your API queries to that company. If you expect to build a commercial program, or you expect to make extensive uses of their APIs, you may have to pay a fee to the company for the use of the API key.

The process of obtaining an API key from a company is typically very easy. An

API key can be obtained from Alpha Vantage at this site: www.alphavantage.co/support/#api-key. There are only a few questions to answer in the form, as shown in Figure 10-3.

Figure 10-3. Requesting an API key from Alpha Vantage

If you fill out the form and press the GET FREE API KEY button, you should soon get an e-mail back that includes a string of about 16 characters. This is your API key. You can use that key to make API requests to Alpha Vantage.

As the final piece of information, you need to add the following:

`&apikey=<yourAPIKeyHere>`

290

Chapter 10 Internet Data

Let's put it all together. In order to get the stock price of Apple, you would build up this URL:

`https://www.alphavantage.co/query?
function=BATCH_STOCK_QUOTES&symbols=AA`

`PL&apikey=xxxxxxx`

The xxxxxx is the API key you received from AlphaVantage.

Example Program to Get Stock Price Information

Using an API

Now that we have all the pieces, we can build a program to get the price of a stock using an API. If we make a call using the API just discussed, with a valid API key, the answer comes back as a single string that looks like this: {

```
"Meta Data": {  
  "1. Information": "Batch Stock Market Quotes",  
  "2. Notes": "IEX Real-Time Price provided for free by IEX  
(https://iextrading.com/developer/).",  
  "3. Time Zone": "US/Eastern"  
},  
"Stock Quotes": [  
  {  
    "1. symbol": "AAPL",  
    "2. price": "177.8500",  
    "3. volume": "20536464",  
    "4. timestamp": "2018-04-18 16:30:20"  
  }  
]  
}
```

In the next chapter, I show you how this information is laid out in a special text format (called JSON). For now, let's just view this information as one long string. We can see that the stock price for today is 177.8500. But in order to find the price of a stock in this string generically, we will have to write code to build a slice to extract it. In the preceding string, we see that the price is preceded by

the string "2. price": and is ended by a double quotation mark. We'll write some code to calculate the indices needed to identify that slice.

291

Chapter 10 Internet Data

Here is our full program to make the request, get the response, and extract the price information, based on any stock symbol entered by the user: # Getting a stock quote

```
import urllib.request

API_KEY = 'xxxxxx' ## <- Replace xxxxxx with your API key

# Data provided for free by Alpha Vantage. Website: alphavantage.co

#
# typical URL:
# https://www.alphavantage.co/query?function=BATCH_STOCK_QUOTES&
# symbols=AAPL&apikey=<key>

def getStockData(symbol):
    baseURL = 'https://www.alphavantage.co/query?function=BATCH_STOCK_
    QUOTES&symbols='
    ending = '&apikey=' + API_KEY
    fullURL = baseURL + symbol + ending
    print()
    print('Sending URL:', fullURL)
    # open the URL
    connection = urllib.request.urlopen(fullURL)
```

```

# read and convert bytes to a string
responseString = connection.read().decode()

print('Response is: ', responseString)

# Look for a prefix in the response
prefixString = '"2. price": "'

# do a little math to figure out the start and end index of the real price:
prefixStringPosition = responseString.index(prefixString)
prefixStringLength = len(prefixString)
start = prefixStringPosition + prefixStringLength
end = responseString.index('"', start)

```

292

Chapter 10 Internet Data

```

# extract the price using a slice, and return it
price = responseString[start:end]

return price

while True:

    print()

    userSymbol = input('Enter a stock symbol (or press ENTER to quit): ')

    if userSymbol == "":

        break

    thisStockPrice = getStockData(userSymbol)

```

```
print()
```

```
print('The current price of', userSymbol, 'is:', thisStockPrice)
```

```
print()
```

```
print('OK bye')
```

This program's code is very straightforward. At the bottom, the main code has an infinite loop where we ask the user for a stock symbol. That symbol is passed to the `getStockData` function. That function builds a full URL including the operation we want to perform, the stock symbol, and our API key. The function makes a request over the Internet using that URL. The server hands back a long string as documented earlier.

(I've left in the print statements that show the URL that is sent and the response string that is returned—you can comment these lines out or remove them if you want.) The function then extracts the price of the stock by calculating the start and end indices for the appropriate slice. Finally, it returns that price to the caller.

An important point here is that companies build APIs like this so that programs

running on computers and devices can quickly get the data they are asking for. Some APIs are intended only for the use of the company's employees. Others, like the Alpha Vantage's stock API, are available to the general public. Unlike using the screen-scraping technique, where the screen layout may change at any time, APIs are designed not to change, although as a company learns how its users (programmers) are using its data, it may choose to amend some API details and/or add new API calls.

293

Chapter 10 Internet Data

Caution APIs have the potential for abuse. If you wind up making too many calls to an API per hour, or if you make calls too quickly, you may be locked out from making such calls. Owners of sites that have APIs often *throttle* the number of calls allowable within a certain time period. please do not abuse APIs with your

code.

Example Program to Get Weather Information

There is a wonderful site at OpenWeatherMap.org that allows programmers to retrieve a wide variety of weather data from around the world. Its APIs are well documented at <http://openweathermap.org/API>. In order to use any of the APIs, you must first obtain a free API key using a similar form to the one you saw earlier.

There are many choices for the types and quantities of weather information you can retrieve. As a demonstration, I'll show you how to get the current weather information for any city. Our more specific goal is to retrieve the current temperature in that city.

The API works as follows. You make a call to the base API and, as data, supply the city you want information about, the return format for your data, and your API key. For example: `api.openweathermap.org/data/2.5/weather?q=Phoenix&mode=xml&APPID=xxxxxx` Using that URL, here is an example of the current weather information returned for the city of Phoenix: `<?xml version="1.0" encoding="UTF-8"?>`

```
<current><city id="5308655" name="Phoenix"><coord lon="-112.08"
```

```
lat="33.45"></coord><country>US</country><sun rise="2018-04-20T12:50:59"
```

```
set="2018-04-21T02:03:33"></sun></city><temperature value="292.37"
```

```
min="291.15" max="293.15" unit="kelvin"></temperature><humidity value="22" unit="%"></humidity><pressure value="1016" unit="hPa"></pressure><wind><speed value="4.6" name="Gentle Breeze"></speed><gusts value="9.3"></gusts><direction value="270" code="W" name="West"></direction></wind><clouds value="20" name="few clouds">
```

```
</clouds><visibility value="16093"></visibility><precipitation mode="no">
```

```
</precipitation><weather number="801" value="few clouds" icon="02d">
```

```
</weather><lastupdate value="2018-04-20T17:58:00"></lastupdate></current>
```

704

Chapter 10 Internet Data

There is a lot of information there, and it may seem intimidating. In the next chapter, I show you how this information is laid out in another text format called XML. For now, let's just view this as one long string. If you look through the string, you will see a tag that says `<temperature value=`". Immediately following that is the actual temperature.

We can use the same approach we used for getting stock data. That is, we can calculate the indices of the start and end points of the temperature and use a slice to extract the information we want.

However, the data this program generated for the temperature in Phoenix shows a

value of 292.37, which seems quite hot, even for Phoenix. In the United States, we use the Fahrenheit scale, and most of the rest of the world uses the Centigrade scale, so when reporting temperatures, OpenWeatherMap.org decided to represent temperatures in yet a third scale: Kelvin. The Kelvin scale is based on the concept of absolute zero. To make the answers clearer for readers in the United States, I wrote and used a small function to convert a temperature from degrees Kelvin into degrees Fahrenheit. The resulting Fahrenheit temperature is a much more enjoyable 66.866 degrees.

The following full program allows the user to enter the name of any major city. The program makes an API call to get all the weather information for that city. After the server responds, the program then extracts the temperature information and converts it to a Fahrenheit value that it reports to the user: `import urllib.request`

```
# API documentation from: http://openweathermap.org/API
```

```
# Sample, try: api.openweathermap.org/data/2.5/weather?  
q=Phoenix&mode=xml& APPID=xxxxx
```

```
API_KEY = 'xxxxx' ## <- Replace xxxxx with your API key
```

```
def getInfo(city):
```

```
URL = 'http://api.openweathermap.org/data/2.5/weather?q=' + city +  
'&mode=xml'+ '&APPID=' + API_KEY  
print("URL request is: " + URL)  
print()  
# Make the request and save the response as a string.  
connection = urllib.request.urlopen(URL)  
responseString = connection.read().decode()
```

295

Chapter 10 Internet Data

```
print(responseString)  
print()  
prefixString = '<temperature value="'  
# do some small math to figure out the start and end index of the  
temperature:  
prefixStringLength = len(prefixString)  
prefixStringPos = responseString.index(prefixString)  
start = prefixStringPos + prefixStringLength  
end = responseString.index('"', start)  
# extract the temperature and return it  
degreesK = responseString[start : end] # this is in degrees Kelvin  
degreesK = float(degreesK)
```

```

return degreesK

# Convert from Kelvin degrees to Fahrenheit

def convertKToF(degreesK):
    degreesF = (1.8 * (degreesK - 273.)) + 32
    return degreesF

while True:
    city = input('What city would you like the temperature of? ')
    if city == "":
        break

    tempK = getInfo(city)

    # Convert from Kelvin degrees to Fahrenheit

    tempF = convertKToF(tempK)

    print(tempF)

    print()

    print('Bye')

```

296

Chapter 10 Internet Data

This program includes print statements to show the URL that was created and the

response string that was returned. If you comment those lines out, a typical run would look like this:

What city would you like the temperature of? Phoenix

68.990000000000002

What city would you like the temperature of? Boston

38.714000000000034

URL Encoding

When we use parameter values in conjunction with API calls, the values that are passed in are always strings. For most strings made up of standard characters, everything works fine, but certain characters are considered “unsafe” when used in parameter values.

Most importantly, the *space* character is considered not to be safe. The original reason has to do with people reading values from one place and typing them into fields or forms that wind up in URLs. Because the space character is essentially an invisible character, the number of spaces that were in the original text may not be clear.

In order to include a space in a parameter value, the space character must be translated to either the plus character (+) or the numeric value of the space character.

Every character is assigned a unique number. All characters can be represented by a special string that gives the number associated with that character as a hexadecimal (base 16) number. For example, the space character as a hexadecimal number is written as %20. The process of replacing a character with another character or sequence of characters is called *encoding*.

If we wanted to only encode the space character, we could take any string that we might use as a parameter value in a URL and apply a string replace operation to it. For example: >>> originalString = 'New Jersey'

```
>>> encodedString = originalString.replace(' ', '+')
```

```
>>> print(encodedString)
```

New+Jersey

```
>>>
```

Chapter 10 Internet Data

Or this:

```
>>> originalString = 'New Jersey'
>>> encodedString = originalString.replace(' ', '%20')
>>> print(encodedString)

New%20Jersey
>>>
```

Either of these versions of the encoded string could then be used within a URL in an API call. When a parameter value is received by a server, any plus character or %20

sequence is decoded back to the space character.

There are a number of other characters that are also considered unsafe for use in a URL, including the following:

- " (Quote mark)

- < and > (Less-than and greater-than symbols)
- # (Pound sign)
- % (Percent sign)
- And the following: {, }, |, \, ^, ~, [,], `

That is a lot of characters to remember and find replacement hexadecimal

representations for. Fortunately, there is a built-in Python function that can do this work for us. If we ever believe that a value of a parameter to be passed in a URL might contain any of these characters, then before building the value into a URL, we can use the following function from the urllib package: `import urllib.parse`

```
encodedString = urllib.parse.quote_plus(<original string>)
```

To use it, you pass in the original string, and it returns an encoded version of the string that works within a URL. For example, if we want to make an API call where we want to specify a value of the string 'New Jersey', we would encode it this way: >>> import url.libparse

```
>>> originalString = 'New Jersey'
```

```
>>> encodedString = urllib.parse.quote_plus(originalString)
```

```
>>> print(encodedString)
```

```
New+Jersey
```

```
>>>
```

```
298
```

Chapter 10 Internet Data

This call encodes the space as a plus in the same way you saw earlier with the string replace operation. However, the call to `urllib.parse.quote_plus` takes care of encoding *all* potentially unsafe characters for us. Here is an example: >>> import url.libparse

```
>>> originalString = 'The sales tax of "New Jersey" is > 1%'
```

```
>>> encodedString = urllib.parse.quote_plus(originalString)
```

```
>>> print(encodedString)
```

```
The+sales+tax+of+%22New+Jersey%22+is+%3E+1%25
```

```
>>>
```

As a result of this call, in addition to encoding the spaces into plus signs, the double-quote characters have been converted to their hexadecimal form (%22), the greater-than character has been changed to a %3E, and the percent sign has been replaced with %25.

After doing this type of encoding using `urllib.parse.quote_plus`, you can be assured that your parameter values are safe for transmission to a server within a full URL.

Summary

This chapter was all about getting text data over the Internet. I explained the request/

response model, which is used to exchange information between a computer and a

server. I then showed you how to take a request and add parameter values by adding them in as name/value pairs to the end of a URL string. After reading the response and decoding it, the response string is typically saved in a Python variable. I demonstrated this technique by showing you how to get a stock price from a financial site. We built a Python program that made the same request that we made in a browser, and got back the same HTML the browser got back. Then we extracted the stock price we were looking for. Although this was an interesting demonstration, it is not the proper way to get information because the HTML is designed for display on the screen.

The proper way to get data over the Internet is to use an API. Using an API allows us to get the data we are looking for in a much more concise way. Many companies attempt to ensure that their APIs are not overused or used maliciously by issuing and requiring the use of an API key. I explained how to build a URL (for an API) that includes a base URL and add parameters onto the end. Using an API, I showed you how to get the price of a given stock.

I also showed a program that uses an API to get weather information for any given city.

The chapter wrapped up by discussing a technique used to ensure that potentially unsafe characters can be encoded so that they are correctly transmitted in requests.

Data Structures

Let's start this chapter with a definition.

Definition A

data structure is a collection of multiple pieces of data, arranged in a way that the data can be accessed efficiently.

The only data structure we have discussed so far is a list. A list allows us to refer to any one of multiple pieces of data using an index. Python has a few more built-in data structures.

This chapter covers the following topics:

- Tuples
- Lists of lists
- Representing a grid or a spreadsheet
- Representing the world of an adventure game
- Reading a comma-separated value (.csv) file
- Dictionary
- Using the in operator on a dictionary
- Programming challenge
- A Python dictionary to represent a programming dictionary
- Iterating through a dictionary
- Combining lists and dictionaries
- JSON: JavaScript Object Notation
- Example program to get weather data

I. Kalb, *Learn to Program with Python 3*, https://doi.org/10.1007/978-1-4842-3879-0_11

Chapter 11 Data Structures

- XML data
- Accessing repeating groupings in JSON and XML

Tuples

Python has a built-in data structure called a tuple. (There is ongoing debate about whether this should be pronounced “toople” or “tuhpple.” The latter, pronounced as in “quintuple,” seems more popular, but both are acceptable.) A *tuple* is essentially a list that cannot be changed. We’ll review some basic operations of a list and then look at how a tuple differs from it. Let’s start by creating a list: >>> friendsList = ['Joe', 'Martha', 'John', 'Susan']

```
>>> print(friendsList)
```

```
['Joe', 'Martha', 'John', 'Susan']
```

```
>>>
```

We can find the length of the list using the len function and access any element in this list using an index: >>> print(len(friendsList))

```
4
```

```
>>> print(friendsList[0])
```

```
Joe
```

```
>>> print(friendsList[3])
```

```
Susan
```

```
...
```

```
>>>
```

We can also change the value of a given element in the list and add (append) an element to the list:

```
>>> friendsList[2] = 'Greg'
```

```
>>> print(friendsList)
```

```
['Joe', 'Martha', 'Greg', 'Susan']
```

```
>>> friendsList.append('Diane')
```

```
>>> print(friendsList)
```

```
['Joe', 'Martha', 'Greg', 'Susan', 'Diane']
```

```
>>>
```

302

Chapter 11 DATA Structures

We can set a new value and perform the append operation because lists are mutable (changeable). By contrast, a tuple is immutable (not changeable).

A tuple is defined using a similar, but slightly different syntax from a list. Instead of the left and right square brackets used to define a list, a tuple is defined using left and right parentheses: (<element1>, <element2>, ... <elementN>)

Like a list, a tuple is typically created in an assignment statement:

<tupleVariable> = (<element1>, <element2>, ... <elementN>) Let's say we want to create a list of friends. If we know that the list of friends will not change during a run of the program, we would create a tuple of friends and initialize it at the start of the program, as follows: >>> friendsTuple = ('Joe', 'Martha', 'John', 'Susan')

```
>>> print(friendsTuple)
```

```
('Joe', 'Martha', 'John', 'Susan')
```

```
>>>
```

So far, it looks the same as our earlier friendsList, except for the use of parentheses instead of square brackets. We can use the len function to see how many elements are in a tuple and the bracket syntax to get at an individual element of a tuple: >>> print(len(friendsTuple))

```
4
```

```
>>> print(friendsTuple[0])
```

```
Joe
```

```
>>> print(friendsTuple[3])
```

```
Susan
```

```
>>>
```

But if we try to modify an individual element of the friendsTuple, we get an error message:

```
>>> friendsTuple[2] = 'George'
```

Traceback (most recent call last):

File "<pyshell#19>", line 1, in <module>

```
303
```

Chapter 11 Data Structures

```
friendsTuple[2] = 'George'
```

```
TypeError: 'tuple' object does not support item assignment
```

```
>>>
```

If we try to append a new name onto the tuple, we also get an error message:

```
>>> friendsTuple.append('Diane')
```

Traceback (most recent call last):

File "<pyshell#21>", line 1, in <module>

```
friendsTuple.append('Diane')
```

AttributeError: 'tuple' object has no attribute 'append'

```
>>>
```

Both of these error messages show that the contents of a tuple cannot be changed.

You might be asking, “What good is this? Why would I ever want to use a tuple over a list?” The answer is speed. When a list is represented as a tuple, Python internally organizes the data in a way that it can access each individual element faster than in a list. Therefore, if you want to write code that runs as fast as possible, then look for any case where you have a list that never changes in your program. You can redefine it from a list to a tuple by changing the square brackets to parentheses. Eventually, this concept becomes second nature. You start thinking of unchanging lists as tuples and define them that way right from the start.

There is one additional small benefit to using a tuple. If you have a list of data and you want to ensure that there is no code that makes any changes to it, use a tuple.

Any code that attempts to append to, delete from, or modify an element of a tuple will generate an error message. The offending code can quickly be identified and corrected.

Note If you ever write any code using pyGame (an extension to python that allows you to put graphics on the screen), you will notice that screen coordinates are almost always written as x, y tuples, and in (<xValue>, <yValue>). Further, rectangles in pyGame are typically written as four-element tuples: (<xValue>, <yValue>, <width>, <height>).

Chapter 11 Data Structures

Lists of Lists

In our earlier discussion of lists, I said that one interesting thing about a list is that the content—the data inside a list—can be of any data type. It turns out that not only can the data be of type integer, float, string, or Boolean, but any element of a list can also be a list.

For example, consider the following list:

```
>>> myList = [5, -1, [23, 45, 14], 62]
```

```
>>> print(myList)
```

```
[5, -1, [23, 45, 14], 62]
```

To find out how many elements are in this list, we'll use the len function:

```
>>> print(len(myList))
```

```
4
```

The list has four elements, but element 2 is also a list:

```
>>> print(myList[2])
```

```
[23, 45, 14]
```

```
>>>
```

If we wanted to get to a value in this list within a list, there are two approaches. First, we could assign the inner list to a new list variable, and then reference the particular element we want from that list: >>>

```
>>> innerList = myList[2]
```

```
>>> print(innerList)
```

```
[23, 45, 14]
```

```
>>> print(innerList[1])
```

```
print(myList[1])
```

```
45
```

```
>>>
```

Or we could use a different syntax. To get to an element of a list within a list, we can do this:

```
<outerList>[<outerListIndex>][<innerListIndex>]
```

```
305
```

Chapter 11 Data Structures

For example:

```
>>> myList = [5, -1, [23, 45, 14], 62]
```

```
>>> print(myList[2][1])
```

```
45
```

```
>>>
```

This syntax reaches into `myList` and gets element 2 (which is the inner list of `[23, 45, 14]`). Because that is a list, we then get element 1 of that list (which is the value 45) and print it. (As you will soon see, this concept can extend to lists of lists of lists, and so on.) **Representing a Grid or a Spreadsheet** Lists within lists are a great way to represent data in a grid or a spreadsheet, or any application where you have a need for rows and columns. Grids can be used to represent the playing boards of many games. For example, we could represent a tic-tac-toe board as a grid of three rows and three columns, like this: `EMPTY = "`

```
X = 'x'
```

```
O = 'o'
```

```
# Build a 3 by 3 grid
```

```
grid = [\
```

```
[EMPTY, EMPTY, EMPTY],\
```

```
[EMPTY, EMPTY, EMPTY],\
```

```
[EMPTY, EMPTY, EMPTY]\
```

```
]
```

As each player makes a move in the game, we would write code to put an X or an O

into the appropriate spot in the grid. For example, if a player decided to place an X in the upper right-hand square, we would use this code to modify that cell: # Typically set the row and col based on user input, this is just for

demonstration:

```
row = 0
```

```
col = 2
```

```
grid[row][col] = X
```

306

Chapter 11 Data Structures

Any game board that is made up of any number of rows and columns can be represented this way. For example, you could build an eight-by-eight grid to represent the board for a game of checkers or chess.

Representing the World of an Adventure Game

Adventure games are a very popular form of text-based games. In an adventure game, the user is placed in a world that can also be represented as a grid. Here is an example of a start to a program that builds a six-by-six grid: # Adventure game demo

```
import random
```

```
EMPTY = ' '
```



```

EMPTY = 'e'

TREASURE = 't'

MONSTER = 'm'

# Build 6 by 6 grid

NROWS_IN_GRID = 6

NCOLS_IN_GRID = 6

grid = [

[EMPTY, TREASURE, EMPTY, EMPTY, EMPTY, MONSTER],\

[EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY],\

[EMPTY, EMPTY, EMPTY, EMPTY, MONSTER, EMPTY],\

[EMPTY, MONSTER, EMPTY, EMPTY, EMPTY, EMPTY],\

[EMPTY, EMPTY, EMPTY, EMPTY, TREASURE, EMPTY],\

[EMPTY, TREASURE, EMPTY, EMPTY, EMPTY, EMPTY],\

]

# Find a random starting cell that is empty

while True:

locRow = random.randrange(NROWS_IN_GRID)

locCol = random.randrange(NCOLS_IN_GRID)

if grid[locRow][locCol] == EMPTY:

break # found an empty cell, we will place the player here

print('Starting at row:', locRow, ' col:', locCol)

```

```
print()
```

```
307
```

```
Chapter 11 DAtA StruCtureS
```

```
while True: # move around the grid
```

```
direction = input('Press L, U, R, or D to move: ')
```

```
direction = direction.lower()
```

```
print()
```

```
if direction == 'l':
```

```
locCol = locCol - 1
```

```
elif direction == 'u':
```

```
locRow = locRow - 1
```

```
elif direction == 'r':
```

```
locCol = locCol + 1
```

```
elif direction == 'd':
```

```
locRow = locRow + 1
```

```
else:
```

```
print('Oops - staying where we are ... ')
```

```
foundInCell = grid[locRow][locCol]
```

```
print('Now at row:', locRow, ' col:', locCol, ' cell contains:',
```

```
foundInCell)
```

```
# Add code here to do whatever you want with the contents of the
```

current cell

(e.g., fight, run, pick up, etc.)

This code is a good start for creating and populating the world represented by the grid, and for handling the navigation within it. To make it fun, you would want to add code to handle the interactions between the player and whatever they find as they navigate around in the world. Additionally, it would be important to add code to check and handle the cases of potentially moving off all edges. For example, the user might be in the first column (column 0) and press the l key to say that they want to go left. In a case like this, you could either have some code that gives the user a message saying that they cannot go there, or allow an action like this to wrap around the grid. That is, if the user tries to go off the left edge of the world, they reappear on the right edge in the same row.

A similar thing could be done for an attempt to move off any edge.

308

Chapter 11 Data Structures

The preceding code works fine, but every time the game is played, the grid is laid out the same way. The following is some code that generates a random grid every time the game is played: # Adventure game demo dynamic

```
import random
```

```
# Define some constants for items that will be found in the grid
```

```
EMPTY = 'e'
```

```
TREASURE = 't'
```

```
MONSTER = 'm'
```

```
SWORD = 's'
```

```
POTION = 'p'
```

```
addInToGrid = (TREASURE, TREASURE, TREASURE, MONSTER,  
MONSTER, MONSTER,\
```

SWORD, SWORD, POTION, POTION)

NROWS_IN_GRID = 6

NCOLS_IN_GRID = 8

Find a random cell that is empty

def findEmptyCell(aGrid, nRows, nCols):

while True:

aRow = random.randrange(nRows)

aCol = random.randrange(nCols)

if aGrid[aRow][aCol] == EMPTY:

return aRow, aCol

Build grid, start it off all empty

grid = []

for r in range(0, NROWS_IN_GRID):

aRow = []

for c in range(0, NCOLS_IN_GRID):

aRow.append(EMPTY)

grid.append(aRow)

Add in items randomly

for item in addInToGrid:

locRow, locCol = findEmptyCell(grid, NROWS_IN_GRID, NCOLS_IN_GRID)

grid[locRow][locCol] = item

Chapter 11 DATA Structures

```
# For testing, print the grid, row by row

for thisRow in grid:

    print(thisRow)

    print()

locRow, locCol = findEmptyCell(grid, NROWS_IN_GRID, NCOLS_IN_GRID)

# For testing, print out the starting location so we know where we are in
the grid

print('Starting at row:', locRow, ' col:', locCol)

< ... same navigation code as before >
```

The important difference in this code is that the grid is built dynamically. To do that, we start our grid variable as an empty list. Then we use a for loop to iterate through all the potential rows in the grid. Notice that we are using constants to define the number of rows and columns in the grid, and that we use these constants in our for loops. Each time through our outer loop, we initialize an aRow variable to the empty list. Then we have a nested for loop that appends a value of EMPTY to aRow. At the end of the inner loop, we have built a single row of our grid, represented as a list of all EMPTYs. Each time through the outer loop (for each row), we then append this list to our grid. This ends up building the grid as a list of lists.

Next, we iterate over a tuple of items to be added to the grid that were stored in a variable named addInToGrid. For each item in that tuple, we call findEmptyCell. This function returns both a row and a column of an empty cell in the grid. We use the syntax of two indices in brackets to identify the outer index (the row) and an inner index (the column) of the cell into which we will store a value. The code calls the same function to find a random starting point for the user in the grid.

Just for demonstration purposes, let's print the resulting grid, a row at a time, so that you can see what was built. Here is the output from a typical run: ['e', 'p', 'e', 'e', 't', 'm', 'e', 'e']

['e', 'e', 'e', 'e', 'e', 'p', 'e', 'e']

['e', 'e', 'e', 'e', 'e', 't', 'e', 'e']

['e', 'e', 'e', 'e', 'm', 'm', 's', 'e']

['e', 'e', 'e', 'e', 'e', 'e', 'e', 'e']

['t', 's', 'e', 'e', 'e', 'e', 'e', 'e']

Starting at row: 5 col: 5

Press L, U, R, or D to move:

310

◀	A	B	C	D	E	F	G	H
1	Name	Homework 1	Homework 2	Homework 3	Homework 4	Homework 5	Midterm	Final
2	Joe	20	20	19	20	20	34	45
3	Mariah	20	17	20	20	20	34	52
4	John	20	18		18	16	40	55
5	Mary	20	20	20	16	20	27	48
6	Fred		20	20		20	33	45
7	Martha	20	20	20	20	20	38	58
8	Craig	19	20	20	18	19	40	60
9	Kathy	15		20		20	40	56
10	Miles	20	20	20	20	20	26	58
11	Stacey	17	18	15	16	18	38	44
12	George	20		20	20		25	34
13	Sue	20	20		20	19	27	54
14	Tom	20	20	20	20	20	35	58

Chapter 11 Data Structures

Reading a Comma-Separated Value (.csv) File

Another example of data that can be represented as a list of lists is data that comes from a spreadsheet program. We'll work through an example of how we can take data from a spreadsheet created in Microsoft Excel (and probably other spreadsheet programs) and bring it into a Python program.

Figure [11-1](#) shows a spreadsheet that a teacher might construct for keeping track of grades on homework assignments, a midterm exam, and a final exam.

Figure 11-1. Grades spreadsheet

In this course, the homework assignments have a maximum of 20 points each, the midterm has a maximum of 40 points, and the final has a maximum of 60 points.

Therefore, the maximum total possible points for the class is 200 points. If a student does not turn in an assignment, the cell representing that assignment is left blank.

Spreadsheet files like this are typically saved with the default format and the .xls or .xlsx extension. These are standard file formats for Microsoft Excel.

However, as an option, you can click Save As and choose to save the file as a .csv file, which stands for *comma-separated value*. If you choose to save a spreadsheet as a .csv file, the data is written out line by line in plain text, where the data of each cell is separated from the adjacent one by inserting a comma character. I have saved the spreadsheet shown in Figure [11-1](#) in .csv format; the resulting file looks like what's shown in Figure [11-2](#).

Name	Homework 1	Homework 2	Homework 3	Homework 4	Homework 5	Midterm	Final
Joe	20	20	19	20	20	34	45
Mariah	20	17	20	20	20	34	52
John	20	18	18	16	40	55	
Mary	20	20	20	16	20	27	48
Fred	20	20	20	33	45		
Martha	20	20	20	20	20	38	58
Craig	19	20	20	18	19	40	60
Kathy	15	20	20	40	56		
Miles	20	20	20	20	20	26	58
Stacey	17	18	15	16	18	38	44
George	20	20	20	25	34		
Sue	20	20	20	19	27	54	
Tom	20	20	20	20	20	35	58

Chapter 11 Data Structures

Figure 11-2. *Grades spreadsheet data saved as a comma-separated value file*

The first line contains the titles of the columns. Following that is one line for each student. As you can see, the data values in each of these lines are separated by commas.

Notice also that any cell that was empty is represented as zero characters in the text line—that is, a missing entry is represented by two commas. Now we need a way to read data formatted this way into a Python program.

Definition *Parse* means to take information and separate it into more easily processed components. For example, the python compiler parses the code you write and breaks it down into the individual words and symbols in each line, so

that it can turn your code into the bytecode form that can run on the computer.

To help read in and parse the data, let's use another of Python's many built-in packages. There is a package (not surprisingly, called the csv package) that is designed to read in CSV-formatted files.

312

Chapter 11 Data Structures

In the following code, we read in the data from this .csv file, calculate a score for each student, and then translate that score into a letter grade (the translation to a letter grade comes from code that we developed earlier with the if/elif/else statements): # Read grades from csv file, compute grade letter for course

```
import csv # Comma separated value package
```

```
DATA_FILE_NAME = 'GradesExample.csv'
```

```
#Convert a number score to a letter grade:
```

```
def letterGrade(score):
```

```
    if score < 60:
```

```

if score >= 90:

    letter = 'A'

elif score >= 80:

    letter = 'B'

elif score >= 70:

    letter = 'C'

elif score >= 60:

    letter = 'D'

else:

    letter = 'F' #fall through or default case

return letter

# Open the file in 'read Universal' (return char) mode)

# This allows for dealing with files created by spreadsheet programs like Excel

fileHandle = open(DATA_FILE_NAME, 'rU')

# Let the csv reader parse the file into rows

csvParsed = csv.reader(fileHandle)

# Treat each row (which represents data for a single student) as a list

readingHeaderLine = True

for row in csvParsed: # iterate through each line

    if readingHeaderLine: # first line?

        readingHeaderLine = False

```

```
continue # skip the header line
```

```
313
```

```
Chapter 11 DATA Structures
```

```
# This is what the data looks like coming in to the program
```

```
#print('Original: ', row)
```

```
name = row[0] # save the student's name
```

```
total = 0 # prepare to add 'em up
```

```
for index in range(1, 8): # elements 1 through 7 are the scores
```

```
thisGrade = row[index]
```

```
if thisGrade == ":
```

```
thisGrade = 0.0 # change a nothing to a zero
```

```
else:
```

```
thisGrade = float(thisGrade) # convert score from string to
```

```
float
```

```
total = total + thisGrade
```

```
percent = (total * 100.) / 200. # out of a possible 200 points
```

```
gradeToReport = letterGrade(percent)
```

```
print(name, ' Percent:', percent, ' Letter Grade:', gradeToReport)
```

```
fileHandle.close() #close the file
```

This code starts by importing the csv package. In the main code, we open the file in a new way, specifying the open mode as 'rU', which stands for *read universal*. This mode allows programs to read text files that were created on any operating

system, because these files may have a variety of different end-of-line and/or newline characters. Once the file is opened, we call `csv.reader`. This is an operation in the `csv` package that reads through the entire file and modifies the data so that each line of the file is represented as a Python list. (Internally, it most likely calls the Python `split` function to separate the individual pieces of data.) When that completes, our code goes through a loop, iterating for each row in the original file.

We set a `readingHeaderLine` Boolean variable to `True` before the loop started.

Inside the loop, we treat each row as a list. The first row is a list containing the header information (Name, Homework1, Homework2, and so on). We don't want to do anything with this line, so all we do is set the `readingHeaderLine` Boolean to `False` to indicate that we are no longer looking at the header line. Then we use a `continue` statement to send control back to the top of the loop.

314

Chapter 11 Data Structures

For each subsequent row, we now can deal with the data representing a single

student. That data is made up of a list of eight elements: element 0 is the student's name, and elements 1 through 7 are the score values, where each is a string. We build another loop to add up the scores of all homework values and the two test scores. If we find that a value is missing (which would come into the program as an empty string), then we give the student a zero for that score. When we are finished with that loop, we have the total score for that student. We then call our `letterGrade` function to convert the score into a letter grade. As the last thing in our loop, we write out this student's name, percentage, and letter grade. When we are done, we close the file.

Running the program generates the following output:

Joe Percent: 89.0 Letter Grade: B

Mariah Percent: 91.5 Letter Grade: A

John Percent: 83.5 Letter Grade: B

Mary Percent: 85.5 Letter Grade: B

Martha Percent: 98.0 Letter Grade: A

Fred Percent: 69.0 Letter Grade: D

Martha Percent: 98.0 Letter Grade: A

Craig Percent: 98.0 Letter Grade: A

Kathy Percent: 75.5 Letter Grade: C

Miles Percent: 92.0 Letter Grade: A

Stacey Percent: 83.0 Letter Grade: B

George Percent: 59.5 Letter Grade: F

Sue Percent: 80.0 Letter Grade: B

Tom Percent: 96.5 Letter Grade: A

If we want to, we could easily modify the code to write a new .csv file where each student line could contain the existing information and the additional percent and/or letter grade information (separated by commas). The resulting .csv file could then be opened in a spreadsheet program, such as Microsoft Excel.

Also, if we had wanted to do more analysis of the scores data—for example, ranking students or scores per assignment or test—we could have saved all the information in a larger list of lists data structure. In other words, we could have created an empty list, like this: `allScores = []`

315

Chapter 11 Data Structures

Then every time we iterated through the loop, we could have appended the current row of data into that list, like this: `allScores.append(row)`

With this approach, we can have all the data from the original .csv file in a single Python list of lists, and we can do any analysis we want.

Dictionary

Another extremely important data structure available in Python is called a dictionary. A *dictionary* is similar to a list in that it allows you to refer to a collection of data by a single variable name. However, it differs from a list in one fundamental way. In a list, order is important, and the order of the elements in a list never changes (unless you explicitly do so). Because the order of elements in a list is important, you refer to each element in a list using its index (its position within the list).

In a dictionary, the data is represented in what are called *key/value* pairs. The syntax of a dictionary looks like this: {<key>:<value>, <key>:<value>, ..., <key>:<value>}

Note that this is the only place in Python where the curly braces { and } are used.

I'll show you an example of how this works. Imagine that we wanted to represent

several attributes or properties of a physical object using Python. We could create a single variable for each item. For example, let's try to describe a house using several variables: color = 'blue'

```
style = 'colonial'
```

```
numberOfBedrooms = 4
```

```
garage = True
```

```
burglarAlarm = False
```

```
streetNumber = 123
```

```
streetName = 'Any Street'
```

```
city = 'Anytown'
```

```
state = 'CA'
```

```
price = 625000
```

Chapter 11 Data Structures

Variables like these work fine. But the data in these variables is all related—each variable is a property of a single house. We could build a dictionary to represent the related data about the house. The same information built as a dictionary would look like this: `houseDict = {'color' : 'blue', 'style' : 'colonial', 'numberOfBedrooms' : 4,`

`'garage' : True, 'burglarAlarm' : False, 'streetNumber' : 123,`

`'streetName' : 'Any Street', 'city' : 'Anytown', 'state' : 'CA',`

`'price' : 625000}`

We are naming this dictionary `houseDict` to make it clear that this is a dictionary.

Again, this is not a requirement; we are using a name like this as an extension to our naming convention. In this example, all the keys of this dictionary are strings, which is a very common practice. However, the keys in a dictionary can be of any type of immutable data—integers, floats, Booleans, and tuples can also be used as keys.

Whatever data type you use for keys, each key in a dictionary must be unique. The values in a dictionary can be of any type.

Let's print out `houseDict` to show that Python understands the dictionary data structure:

```
print(houseDict)
```

```
{'color': 'blue', 'style': 'colonial', 'numberOfBedrooms': 4, 'garage': True,
```

```
'burglarAlarm': False, 'streetNumber': 123, 'streetName': 'Any Street',
```

```
'city': 'Anytown', 'state': 'CA', 'price': 625000}
```

Dictionaries rely on the key/value pair relationships rather than on positioning.

Therefore, when we want to access any piece of data in a dictionary, we do it by using a key as an index (rather than the position index that we use with a list). Here are some examples: >>> print(houseDict['color'])

blue

>>> print(houseDict['state'])

CA

>>> print(houseDict['numberOfBedrooms'])

4

>>>

317

Chapter 11 Data Structures

To assign a new value for an existing key in a dictionary, we use an assignment statement, like this:

>>> houseDict['price'] = 575000 #change value of an existing key

>>> print(houseDict)

{'color': 'blue', 'style': 'colonial', 'numberOfBedrooms': 4, 'garage': True,

'burglarAlarm': False, 'streetNumber': 123, 'streetName': 'Any Street',

'city': 'Anytown', 'state': 'CA', 'price': 575000}

>>>

To add a new key/value pair into a dictionary, we use an assignment statement the same way. If the key we are specifying does not exist in the dictionary, then the key/value pair is added to the dictionary: >>>

houseDict['numberOfBathrooms'] = 2.5 # numberOfBathrooms is not in the dictionary yet


```
>>> print(houseDict)

{'color': 'blue', 'style': 'colonial', 'numberOfBedrooms': 4, 'garage': True,
'burglarAlarm': False, 'streetNumber': 123, 'streetName': 'Any Street',
'city': 'Anytown', 'state': 'CA', 'price': 575000, 'numberOfBathrooms': 2.5}

>>>
```

Notice that the `numberOfBathrooms` key has been added to the dictionary.

There are two additional operations (functions) you can use on a dictionary. If you want, you can find all the keys defined in a dictionary with a call to `<dictionary>`.

`keys()`. You can find all the values with a call to `<dictionary>.values()`. Both calls return an iterable list—perfect for use in a `for` statement. Here is an example using our previously defined dictionary: `>>> print(houseDict.keys())`

```
dict_keys(['color', 'style', 'numberOfBedrooms', 'garage',
'burglarAlarm', 'streetNumber', 'streetName', 'city', 'state', 'price',
'numberOfBathrooms'])
```

```
>>> print(houseDict.values())

dict_values(['blue', 'colonial', 4, True, False, 123, 'Any Street',
'Anytown', 'CA', 575000, 2.5])

>>>
```

318

Chapter 11 Data Structures

Using the `in` Operator on a Dictionary

When we try to access an element in a list, we need to ensure that any index we

use is a valid number. That is, the index has to have a value between 0 and the length of the list minus 1. (Remember that if a list has N elements, then the valid indices are 0 to $N - 1$.) When accessing items in a dictionary, we have to ensure that we are using a valid key; that is, we have to use a key that exists in the dictionary. If the key we use is in the dictionary, we get the value associated with that key. But if we try to use a key that is not in the dictionary, we get an error: >>> print(houseDict)

```
{'color': 'blue', 'style': 'colonial', 'numberOfBedrooms': 4, 'garage':
```

```
True, 'burglarAlarm': False, 'streetNumber': 123, 'streetName':
```

```
'Any Street', 'city': 'Anytown', 'state': 'CA', 'price': 575000,
```

```
'numberOfBathrooms': 2.5}
```

```
>>>
```

```
>>> print(houseDict['streetName'])
```

```
Any Street
```

```
>>>
```

```
>>> print(houseDict['roofType'])
```

```
Traceback (most recent call last):
```

```
File "<pyshell#65>", line 1, in <module>
```

```
print(houseDict['roofType'])
```

```
KeyError: 'roofType'
```

```
>>>
```

This is similar to what happens if we try to use an index that is too large or too small for a list. In that case, we get an “index out of range” error. With a dictionary, we get a `KeyError`, meaning the key does not exist in the dictionary.

To ensure that we are using a valid key, we can use the `in` operator before

attempting to use a key in a dictionary. The in operator is used like this: <key> in <dictionary>

319

Chapter 11 Data Structures

It returns True if the key is found in the dictionary, or False if the key is not found:

```
>>> print('city' in houseDict)
```

True

```
>>> print('roofType' in houseDict)
```

False

```
>>>
```

In any code where we think a key might not be found, it's a good idea to add some defensive coding to check and ensure that the key is in the dictionary before we attempt to use it on the dictionary. Typically, we build this type of check using an if statement: if myKey in myDict: # OK, we can now successfully use myDict[myKey]

else:

The key was not found, print some error message or take some other

action

Sometimes, it may make more logical sense to code the reverse test. We can use not in to test for the key not being in the dictionary: if myKey not in myDict:

The key was not found, do whatever you need to do

Programming Challenge

This challenge asks you to build a dictionary and use keys into that dictionary to extract information. The information is given as a table of state names (keys) and the population of each state (values). The program should allow the user to enter

the population of each state (values). The program should allow the user to enter the name of a state.

If the state is found in the dictionary, then the program should report the population of that state. If the state is not found, then the program should output a message like “Sorry, but we do not have information for that state.” The program should run in a loop, allow the user to enter any number of states, and then exit when the user presses Return (Mac) or Enter (Windows). As of 2018, the data for the 12 states with the highest populations is shown in [Table 11-1](#).

320

Chapter 11 Data Structures

Table 11-1. *U.S. States with Highest Population*

State

Population

California

39776830

texas

28704330

Florida

21312211

New York

19862512

pennsylvania

12823989

Illinois

12768320

Ohio

11694664

Georgia

10545138

North Carolina

10390149

Michigan

9991177

New Jersey

9032872

Virginia

8525660

This is the solution using a dictionary:

Get the population of a given state

statesDict = {

'California':39776830, 'Texas':28704330, 'Florida':21312211,

'New York':19862512,\

'Pennsylvania': 12823989, 'Illinois': 12768320, 'Ohio':11694664,

'Georgia': 10097000,\

'North Carolina': 10390149, 'Michigan':9991177, 'New Jersey': 9032872,

```

'Virginia': 8525660}

while True:

    usersState = input('Enter a state: ')

    if usersState == "":
        break

    if usersState in statesDict:

        population = statesDict[usersState]

        print('The population of', usersState, 'is', population)

    321

Chapter 11 Data Structures

else:

    print('Sorry, but we do not have any information about',

    usersState)

    print()

```

The code for this program is based on a dictionary of state/population key/value pairs. The main loop allows the user to enter a state name. The program tests to see if the given state is in the dictionary by using the in operator. If the state is found, then the program finds the population of that state and reports it. Otherwise, the program says that it does not have any information about that state.

A Python Dictionary to Represent a Programming

Dictionary

Another example of using a dictionary is a program that works as a real dictionary. In this example, it will be a dictionary of a few of the programming

terms introduced in this book. In the following program, programming terms are used as keys, and their matching definitions are specified as values: # Using a dictionary to represent a dictionary of programming terms

```
programmingDict = {  
    'variable': 'A named memory location that holds a value',  
    'loop' : 'A block of code that is repeated until a certain condition is met.',  
    'function' : 'A series of related steps that form a larger task, often  
called from multiple places in a program',  
    'constant' : 'A variable whose value does not change',  
    'Boolean' : 'A data type that can only have values of True or False'}
```

```
while True:
```

```
    print()
```

```
    usersWord = input('Enter a word to look up (or Return to quit): ')
```

```
    if usersWord == ":
```

```
        break
```

```
    if usersWord in programmingDict:
```

```
        322
```

```
Chapter 11 Data Structures
```

```
        definition = programmingDict[usersWord]
```

```
        print('The definition of', usersWord, 'is:')
```

```
        print(definition)
```

```
    else:
```

```

print()

print('The word', usersWord, 'is not in our dictionary.')

yesOrNo = input('Would you like to add a definition for ' +
usersWord + ' (y/n) ')

if yesOrNo.lower() == 'y':

usersDefinition = input('Please give a definition for ' +
usersWord + ': ')

programmingDict[usersWord] = usersDefinition

print('Thanks, got it!')

print('Done.')

```

This example is very similar to the previous challenge. It starts with a dictionary of programming terms, where the words are the keys and the values are the definitions.

However, this program has an additional twist. If the user enters a word that is not in the dictionary, it asks the user if they want to add a definition for the word they entered. If the user chooses to add a definition, the program allows the user to enter the definition, and the key/value pair is added to the dictionary.

Iterating Through a Dictionary

If you need to iterate through all the elements in a dictionary, similar to a list, you can use a for loop. In the case of a dictionary, however, the variable you specify in the for statement is given the value of a key in the dictionary every time through the loop. Here is an example using the earlier dictionary of state populations: >>> statesDict = {

```
'California':39776830, 'Texas':28704330, 'Florida':21312211,
```

```
'New York':19862512,\
```

```
'Illinois':12821354, 'Pennsylvania':12745536, 'Ohio':11536514,
```



```
'Pennsylvania': 12823989, 'Illinois': 12768320, 'Ohio':11694664,  
'Georgia': 10097000,\n'North Carolina': 10390149, 'Michigan':9991177, 'New Jersey': 9032872,  
'Virginia': 8525660}
```

323

Chapter 11 Data Structures

```
>>> for state in statesDict:
```

```
    print(state)
```

California

Texas

Florida

New York

Pennsylvania

Illinois

Ohio

Georgia

North Carolina

Michigan

New Jersey

Virginia

```
>>>
```

You can be assured that using a for loop this way will iterate through every key in the dictionary. If you need each matching value while iterating through a dictionary, you can reach into the dictionary using the current key in the body of the loop. For example: >>> statesDict = {

'California':39776830, 'Texas':28704330, 'Florida':21312211,

'New York':19862512,\

'Pennsylvania': 12823989, 'Illinois': 12768320, 'Ohio':11694664,

'Georgia': 10097000,\

'North Carolina': 10390149, 'Michigan':9991177, 'New Jersey': 9032872,

'Virginia': 8525660}

>>> for state in statesDict:

population = statesDict[state]

print(state, population)

California 39776830

Texas 28704330

Florida 21312211

New York 19862512

Pennsylvania 12823989

324

Chapter 11 Data Structures

Illinois 12768320

Ohio 11694664

Georgia 10097000

```
North Carolina 10390149
```

```
Michigan 9991177
```

```
New Jersey 9032872
```

```
Virginia 8525660
```

```
>>>
```

Combining Lists and Dictionaries

Now you have seen examples of lists of lists and you can build dictionaries of dictionaries. But highly complex data structures can be built by mixing and matching lists and dictionaries. You can have a list of dictionaries or a dictionary where all values are lists. Beyond that, every sublist or subdictionary can also be a dictionary or a list.

Although that may seem very complicated, data structures like this can be extremely useful in representing hierarchical data.

In this first example, we want to represent a number of cars. This data could refer to cars we own, cars we are interested in purchasing, or even cars that are at a used car dealership waiting to be sold: carsList = [\

```
{
```

```
'make':'Toyota', 'model':'Prius', 'year': 2006, 'color':'gold', 'doors':4,
```

```
'leather':False, 'license': 'ABC123', 'mileage': 777777},\
```

```
{
```

```
'make':'Honda', 'model':'Civic', 'year': 2010, 'color':'red', 'doors':2,
```

```
'leather':False, 'license': 'DEF444', 'mileage': 54321},\
```

```
{
```

```
'make':'Ford', 'model':'Fusion', 'year': 2012, 'color':'blue', 'doors':4,
'leather':True, 'license': 'GHI999', 'mileage': 24680},\
    {
'make':'Chevy', 'model':'Volt', 'year': 2015, 'color':'black', 'doors':4,
'leather':False, 'license': 'JKL444', 'mileage': 7890}\
    ]
```

325

Chapter 11 DATA STRUCTURES

In this example, each element in the list is a dictionary. Each dictionary has an identical set of keys. Given this structure, it would be easy to iterate through all the cars, searching for all cars that match a given set of criteria. For example, if we wanted to search through our list of cars and find all cars that have four doors and mileage less than 50,000 miles, we could use the following code: for carDict in carsList:

```
if (carDict['doors'] == 4) and (carDict['mileage'] < 50000):
```

```
print(carDict['make'], carDict['model'], carDict['license'])
```

That would produce the following results:

Ford Fusion GHI999

Chevy Volt JKL444

The following example is slightly more complicated:

```
personalDataDict = {
```

```
'Joe': {'height':73, 'weight': 200, 'sex':'M', 'age':35,
```

```
'allergies':['tree pollen', 'carrots', 'onions']},\
```

```
'Sally':{'height':58, 'weight': 100, 'sex':'F', 'age':32,
```

```

    'Mary': {'height':35, 'weight': 60, 'sex':'F', 'age':7,
'allergies':['bee stings']},\
'John': {'height':36, 'weight': 75, 'sex':'M', 'age':8,
'allergies':['peanuts']},\
'Mary': {'height':35, 'weight': 60, 'sex':'F', 'age':7,
'allergies':[]}\
}

```

In this example, we have a dictionary of people. We use their names as keys. Each person is represented as a dictionary of key/value pairs. But if you look at the allergies key, you can see the value for each person is a list of all things that person is allergic to.

The list can have any number of elements, including zero. We can find the list of allergies for a specific person in this way: `joesData = personalDataDict['Joe']`

```
joesAllergies = joesData['allergies']
```

```
print(joesAllergies)
```

```
326
```

Chapter 11 Data Structures

```
marysData = personalDataDict['Mary']
```

```
marysAllergies = marysData['allergies']
```

```
print(marysAllergies)
```

The code produces this output:

```
['tree pollen', 'carrots', 'onions']
```

```
[]
```

Using a person's name as a key provides a dictionary of information about them.

Within that dictionary, if you use a key of allergies, you get back the list of things that person is allergic to. In the case of Joe, we see that he is allergic to tree pollen, carrots, and onions. Mary's list of things she is allergic to is empty, meaning she is not allergic to anything.

If we wanted to generate a printout of all people and their allergies, we could use the following:

```
for personName in personalDataDict:

    onePersonDict = personalDataDict[personName]

    allergyList = onePersonDict['allergies']

    if allergyList == []:

        print(personName, 'is not allergic to anything')

    else:

        print(personName, 'is allergic to the following:')

        for allergy in allergyList:

            print(' ', allergy)
```

That would produce this output:

John is allergic to the following:

peanuts

Sally is allergic to the following:

bee stings

Joe is allergic to the following:

tree pollen

carrots

onions

Mary is not allergic to anything

327

Chapter 11 Data Structures

Using lists and dictionaries together allows us to build up highly complex data structures. As long as you understand the layers that make up the data structure, Python code can be written in a very straightforward way to get the specific information you want. As shown, extracting the data that is important to you is done using an appropriate sequence of indices and/or keys.

JSON: JavaScript Object Notation

In Chapter [10](#), there is an example of a program that displays stock quote information.

The program asks the user to enter a stock symbol, uses an API call to get current information about the stock, and extracts the stock price from the returned string. As an example, when we created a request for the price of Apple (stock symbol AAPL), the returned string looked like this: {

```
"Meta Data": {
```

```
"1. Information": "Batch Stock Market Quotes",
```

```
"2. Notes": "IEX Real-Time Price provided for free by IEX
```

```
(https://iextrading.com/developer/).",
```

```
"3. Time Zone": "US/Eastern"
```

```
},
```

```
"Stock Quotes": [
```

```
{  
  "1. symbol": "AAPL",  
  "2. price": "165.7500",  
  "3. volume": "65336628",  
  "4. timestamp": "2018-04-20 16:59:43"  
}
```

Now that we have discussed Python's dictionaries, the layout of this string should look more familiar. The information is returned in a special format called JSON (short for *JavaScript Object Notation*). JSON format is a generalized text-based format for structuring data. JSON-formatted data is often used as a mechanism for transmitting hierarchical data in response to requests sent to servers.

328

Chapter 11 Data Structures

The JSON format is almost identical to Python's data structures. In fact, it is so close that there is a Python json package that allows us to translate a string formatted in JSON

into Python lists and dictionaries. If we make the same API call that we did earlier, we can translate the returned string into a Python dictionary using a single call in the json package. Then, rather than using a slice, we can pick out the specific information we are looking for more easily, in a clearer way, and with much less code.

If you look at the preceding response, you will see that the returned string is the equivalent of a Python dictionary. At the top level, there are two key/value pairs. The keys are Meta Data and Stock Quotes. Interestingly, the value associated with the Stock Quotes key is a list. That's because the API allows for more than

one stock symbol in the query. However, our program is only supplying one stock symbol in each request.

In the list of Stock Quotes, each element is a dictionary of information about one stock symbol. We can now easily get the stock price for our stock by asking for the data associated with the key "2. price".

Here is our full program that makes the same request but handles the returned data as JSON-formatted data: # Getting a stock quote

```
import urllib.request

import json

API_KEY = 'xxxxxx' # <- replace this with your API key

# Data provided for free by Alpha Vantage. Website: alphavantage.co

#
# typical URL:
# https://www.alphavantage.co/query?function=BATCH_STOCK_QUOTES&
# symbols=AAPL&apikey=<key>

def getStockData(symbol):

    baseURL = 'https://www.alphavantage.co/query?function=BATCH_STOCK_
    QUOTES&symbols='

    ending = '&apikey=' + API_KEY

    fullURL = baseURL + symbol + ending

    print()

    print('Sending URL:', fullURL)
```

Chapter 11 Data Structures

```
# open the URL

connection = urllib.request.urlopen(fullURL)

# read and convert to a string

responseString = connection.read().decode()

print('Response is: ', responseString)

responseDict = json.loads(responseString) # convert from JSON to a
Python dictionary

print('Response as a dict is:', responseDict)

# The dictionary has an entry with a key of Stock Quotes

stockList = responseDict['Stock Quotes']

# We get back one dictionary for every stock symbol

# Since we only gave 1 stock symbol, we look at element 0

stockDict = stockList[0]

# Reach into the stock dict and pull out the price

price = stockDict['2. price']

return price

while True:

    print()

    userSymbol = input('Enter a stock symbol (or press ENTER to quit): ')

    if userSymbol == "":
```

```
break

thisStockPrice = getStockData(userSymbol)

print()

print('The current price of', userSymbol, 'is:', thisStockPrice)

print()

print('OK bye')
```

The main code of this program is identical to the one introduced in [Chapter 10](#). [All](#)

the changes have been made in the `getStockData` function. In that function, the call to `json.loads` (the `s` means that the input comes from a string) converts the returned JSON

data into its Python form, which in this case is a dictionary. Then, understanding the 330

Chapter 11 Data Structures

structure of the dictionary, we get the value associated with the `Stock Quotes` key. That gives us another list. But because we know that it will only have one element, we can extract element zero, which is a dictionary. Finally, we get the price by using the key 2.

price as a key.

The responses to many different APIs are returned in JSON format. Often the response string is made up of combinations of lists and dictionaries. Applying the techniques demonstrated here allows us to reach into these complex data structures to get the desired information.

Example Program to Get Weather Data

In the previous chapter, we showed a program that retrieved weather information using an API from OpenWeatherMap.org. The OpenWeatherMap API can return

data in a number of different text formats. Here is an example of the result of requesting current weather information for the city of Boston as a JSON string:

```
{ "coord": { "lon": -71.06, "lat": 42.36 }, "weather": [ { "id": 800, "main": "Clear",  
"description": "clear sky", "icon": "01d" } ], "base": "stations", "main": { "temp":  
288.53, "pressure": 1026, "humidity": 13, "temp_min": 286.15, "temp_max": 290.15 },  
"visibility": 16093, "wind": { "speed": 4.1, "deg": 310, "gust": 7.7 }, "clouds":  
{ "all": 1 }, "dt": 1524423120, "sys": { "type": 1, "id": 1296, "message": 0.004,  
"country": "US", "sunrise": 1524390672, "sunset": 1524440080 }, "id": 4930956,  
"name": "Boston", "cod": 200 }
```

In this form, this data may seem rather intimidating. With just a quick glance, you can tell there are a number of nested dictionaries and lists. Let's make this more human readable by adding some newline characters. That should make it easier to understand the overall structure of the data as a dictionary: { "coord": { "lon": -71.06, "lat": 42.36 },

```
"weather": [ { "id": 800,  
"main": "Clear",  
"description": "clear sky",  
"icon": "01d" } ],  
"base": "stations",  
"main": { "temp": 288.53,  
"pressure": 1026,
```

331

Chapter 11 Data Structures

```
"humidity": 13,  
"temp_min": 286.15,
```

```
"temp_max":290.15},  
"visibility":16093,  
"wind":{"speed":4.1,  
"deg":310,  
"gust":7.7},  
"clouds":{"all":1},  
"dt":1524423120,  
"sys":{"type":1,"id":1296,  
"message":0.004,  
"country":"US",  
"sunrise":1524390672,  
"sunset":1524440080},  
"id":4930956,  
"name":"Boston",  
"cod":200}
```

You can see that there is a great deal of weather information available here. To get to the temperature value, we first have to use the main key. The value found there is another dictionary. In that dictionary, you can see the temp key (you also see information about the minimum and maximum temperatures, humidity, pressure, and so on).

The following code is used to get the temperature for any city the user specifies. To run this code, you need to obtain your own API key and assign it to the API_KEY constant.

Note that in the URL that is built up, we are specifying json (as the mode

parameter) to indicate that we want the data to be returned in JSON format: #
Get temperature for a given city

```
import urllib
```

```
import json
```

```
# API documentation from: http://openweathermap.org/API
```

```
# Go to openweathermap.org, get an API Key, and paste it between the quotes  
below'
```

```
API_KEY = 'xxxxxxx' # <- replace with your API Key
```

```
332
```

Chapter 11 Data Structures

```
def getTemperature(city):
```

```
    urlAndParams = 'http://api.openweathermap.org/data/2.5/weather?q=' +  
    city + '&mode=json' + '&APPID=' + API_KEY
```

```
# Make the request and save the response as a string.
```

```
    response = urllib.urlopen(urlAndParams).read()
```

```
    responseDict = json.loads(response) # convert from JSON to a Python  
    dictionary
```

```
    mainDict = responseDict['main'] # get the information associated with  
    the main key
```

```
    degrees = mainDict['temp'] # get the temperature from that dictionary
```

```
    return float(degrees)
```

```
# Convert from Kelvin degrees to Fahrenheit
```

```

def convertKToF(degreesK):
    degreesF = (1.8 * (degreesK - 273.)) + 32
    return degreesF

while True:
    city = input('What city would you like the temperature of? ')
    if city == "":
        break

    tempK = getTemperature(city)
    tempF = convertKToF(tempK)
    print(tempF)
    print()

```

The main code is a loop that continually asks the user to choose a city. It then calls a function called `getTemperature`, passing in the city. `getTemperature` builds the URL

and makes the request. Once we get the returned data back, we convert the response from a JSON-formatted string into a Python dictionary. From there, it is a matter of using the appropriate dictionary keys. First, we use the main key to get the main dictionary.

Within that dictionary, we get the temperature using the `temp` key. (You can see how we could now extract any of the other weather information just as easily.) We convert the temperature string to a float and return it to the main code. Finally, we convert the temperature from Kelvin to Fahrenheit and report the result to the user.

333

Chapter 11 Data Structures

XML Data

In the OpenWeatherMap API we just used, we set the mode parameter to return the data in json format. One of the other options is xml (short for *eXtensible Markup Language*).

In many ways, XML is similar to HTML, the HyperText Markup Language used to define web pages. XML is designed as a self-documenting format that allows computers to exchange data. Similar to HTML, the information to be exchanged is formatted using tags (opening and closing tags). Whereas HTML has a well-defined set of tags that can be used, XML allows you to create your own tags to describe your data.

There are entire books written to explain the intricacies of XML, and I will certainly not try to cover the details here. Instead, I'll show an example of XML-formatted data and explain how to access that data in Python.

If we make the same request to get the temperature of Boston, but we specify the mode as xml, the XML formatted response looks like this: <current><city id="4930956" name="Boston"><coord lon="-71.06"

lat="42.36"></coord><country>US</country><sun rise="2018-04-22T09:51:13"

set="2018-04-22T23:34:40"></sun></city><temperature value="288.53"

min="286.15" max="290.15" unit="kelvin"></temperature><humidity value="13" unit="%"></humidity><pressure value="1026" unit="hPa"></pressure><wind><speed value="4.1" name="Gentle Breeze">

</speed><gusts value="7.7"></gusts><direction value="310" code="NW"

name="Northwest"></direction></wind><clouds value="1" name="clear sky">

</clouds><visibility value="16093"></visibility><precipitation mode="no">

</precipitation><weather number="800" value="clear sky" icon="01d">

</weather><lastupdate value="2018-04-22T18:52:00"></lastupdate></current>

Let's take that XML data and reformat it a little to make it more human readable by adding newline characters and some indenting: <current>

by adding relevant characters and some meaning. Current

<city id="4930956" name="Boston">

<coord lon="-71.06" lat="42.36"></coord>

<country>US</country>

<sun rise="2018-04-22T09:51:13" set="2018-04-22T23:34:40"></sun>

</city>

334

Chapter 11 Data Structures

<temperature value="288.53" min="286.15" max="290.15" unit="kelvin">

</temperature>

<humidity value="13" unit="%"></humidity>

<pressure value="1026" unit="hPa"></pressure>

<wind>

<speed value="4.1" name="Gentle Breeze"></speed>

<gusts value="7.7"></gusts>

<direction value="310" code="NW" name="Northwest"></direction>

</wind>

<clouds value="1" name="clear sky"></clouds>

<visibility value="16093"></visibility>

<precipitation mode="no"></precipitation>

<weather number="800" value="clear sky" icon="01d"></weather>

<lastupdate value="2018-04-22T18:52:00"></lastupdate>

```
<lastupdate value="2010-04-22 11:52:00" /></lastupdate>
```

```
</current>
```

The formatting of the XML data looks very similar to HTML data. Every grouping has start and end tags (for example, `<current>` and `</current>`, `<city>` and `</city>`, and so forth). Each grouping like this is called an *element* or *node*. We'll use the term *node* here so as not to confuse you with elements in a list. Then there are nodes within nodes, each with its own start and end tags. When a node only has text inside it, it is called the *text of the node*. For example, within the city node, there is a country node that looks like this:

```
<country>US</country>
```

Additionally, nodes can have individual name/value pairs. For example, the coord node has values for lon (longitude) and lat (latitude): `<coord lon="-122.08" lat="37.39"></coord>`

In XML, items like these are called *attributes*.

Fortunately, Python provides a package (not surprisingly, called `xml`) that allows programmers to extract the information from XML-formatted data. To use the package, we first have to bring the `xml` package into our programs with an import statement.

Rather than import the entire `xml` package (which includes the ability to write and modify XML documents), we only need the part that allows us to turn XML strings into XML documents and retrieve information. That portion is called `xml.etree`.

335

Chapter 11 DATA STRUCTURES

`ElementTree`. That is a rather long name—Python provides a way to import a package and give it a shorthand name. You do that using the following variation of the import statement: `import <full imported package name> as <shorthand version of package name>` In the following code, we use this variation of the import statement and use a shortened name of `etree`. This is not required, but is often done as a convenience by programmers so that they won't have to type long names when specifying a function in a package.

XML data is often thought of as a tree. Using our example data, try to think of the data in the form of a tree lying on its side. The root of the tree is the node named current. From that node, we see the following nodes: city, temperature, humidity, pressure, wind, clouds, visibility, precipitation, weather, and lastupdate. Coming off of the city node are the coord, country, and sunrise nodes. The coord and sunrise nodes each have two attributes. What we need is a way of taking the data that we have as a string and turning it into a tree-structured document. That is done with a call to a function in the xml package, like this: `tree = etree.fromstring(XMLAsAString)`

After that call, we have the data in a form that allows us to get any information we want to from the XML in its tree form. The xml package has a number of functions that can be used to easily find any individual piece of information we want.

Here is the code for the XML-based version of the program:

```
# Get weather data from openweathermap.org - as XML

import urllib.request

import xml.etree.ElementTree as etree

# API documentation from: http://openweathermap.org/API

API_KEY = 'xxxxxxx' # <- replace with your API Key

def getInfo(city):

    urlWithParams = 'http://api.openweathermap.org/data/2.5/weather?q='\
    + city + '&mode=xml' + '&APPID=' + API_KEY

    336
```

Chapter 11 Data Structures

```
# Make the request and save the response as an XML-formatted string.

connection = urllib.request.urlopen(urlWithParams)
```

```

# Read the data and convert to a string:
responseString = connection.read().decode()

print(responseString)

# Turn the string into an XML document
tree = etree.fromstring(responseString)

# Find the temperature node, then get the value attribute inside it
temperatureInfo = tree.find('temperature')

degrees = temperatureInfo.attrib['value']

return float(degrees)

# Convert from Kelvin degrees to Fahrenheit

def convertKToF(degreesK):

    degreesF = (1.8 * (degreesK - 273.)) + 32

    return degreesF

while True:

    city = input('What city would you like the temperature of? ')

    if city == "":

        break

    tempK = getInfo(city)

    tempF = convertKToF(tempK)

    print(tempF)

    print()

```

The main code and the function to convert from Kelvin are identical to those of the previous example. The changes are in the `getInfo` function. As mentioned earlier, the URL in this version specifies that the mode of the response should be XML. Once we receive the response, we use the following lines to get the specific information we want: # Turn the string into an XML tree

```
tree = etree.fromstring(responseString)
```

337

Chapter 11 Data Structures

```
# Find the temperature tag, then the value attribute inside that
```

```
temperatureInfo = tree.find('temperature')
```

```
degrees = temperatureInfo.attrib['value']
```

The first line converts the response string from the server into an XML tree. The next line reaches into the resulting XML tree structure and finds the temperature group.

Within that group, the last line finds the value attribute. If we wanted to, we could now get to any other piece of data in the XML tree.

There are many more functions available within the `etree` package. More

information on all the functions available to parse XML documents is in the official Python documentation at <https://docs.python.org/3/library/xml.etree>.

[elementtree.html](https://docs.python.org/3/library/xml.etree.elementtree.html).

JSON and XML are two solutions to the same problem of representing arbitrary

hierarchical data. Most often, these two formats are used to transmit data between two computers. JSON is much more succinct and Python-like, or *Pythonic*. JSON is easily accessible in pure Python because JSON-formatted data can be parsed using Python lists and dictionaries. XML is more descriptive because it has tags that identify the data built into the data structure itself. Because of that, equivalent XML-formatted data tends to be considerably longer.

Further, when attempting to read XML-formatted data, code must be written using a set of calls defined in the `xml` package.

Accessing Repeating Groupings in JSON and XML

Sometimes, the data returned by an API has repeating groupings. For example, imagine you made a request to a site to get information about all the members of a band. In response, you get back a set of blocks where each block contains information about one member of the band—for example, the member's name, age, and instrument played.

I'll show what this data structure might look like in JSON format and then in XML format.

I'll also show how we can use Python to access the data about each band member:

Demonstration of repeating blocks in JSON and XML

```
import json
```

```
import xml.etree.ElementTree as etree
```

```
# Build a JSON structure as a triple quoted string
```

```
myJSON = "{
```

```
338
```

```
Chapter 11 DATA Structures
```

```
"bandMembers": [
```

```
{
```

```
"name": "Keith Emerson",
```

```
"age": 32,
```

```
"instrument": "keyboards"
```

```
,
```

```

},
    {
        "name": "Greg Lake",
        "age": 42,
        "instrument": "guitar"
    },
    {
        "name": "Carl Palmer",
        "age": 35,
        "instrument": "drums"
    }
]
}"""

```

```
bandMembersDict = json.loads(myJSON)
```

```
memberList = bandMembersDict['bandMembers']
```

```
for member in memberList:
```

```
    print(member['name'], member['age'], member['instrument'])
```

As you might expect, the repeating blocks are handled as a list.

In the JSON code, the entire structure is converted into a dictionary with only one name/value pair. Using a bandMembers key, we get a list of band members. We then iterate through the list, and each member is represented as a dictionary. For each member, we print out their name, age, and instrument using an appropriate key: # Build an XML structure as a triple quoted string

```
myXML = ""
```

```
<bandMembers>
```

```
<member>
```

```
<name>Keith Emerson</name>
```

```
<age>32</age>
```

```
<instrument>keyboards</instrument>
```

```
339
```

```
Chapter 11 Data Structures
```

```
</member>
```

```
<member>
```

```
<name>Greg Lake</name>
```

```
<age>42</age>
```

```
<instrument>guitar</instrument>
```

```
</member>
```

```
<member>
```

```
<name>Carl Palmer</name>
```

```
<age>35</age>
```

```
<instrument>drums</instrument>
```

```
</member>
```

```
</bandMembers>""
```

```
tree = etree.fromstring(myXML)
```



```
bandMembersList = tree.findall('member')

for member in bandMembersList:

    (print member.find('name').text, member.find('age').text,
     member.find('instrument').text)
```

The XML code is similar. We first convert the structure into an XML tree. We then use a call in the xml package to find all the band members (each is a node). That returns a list of all band members. Like the JSON code, we iterate through all band members. Within each band member, we use the find operation in the xml package to find each member's name, age, and instrument and print out the text associated with each of these nodes.

The output of both of these sections of code is exactly the same:

```
>>>
```

```
Keith Emerson 32 keyboards
```

```
Greg Lake 42 guitar
```

```
Carl Palmer 35 drums
```

```
Keith Emerson 32 keyboards
```

```
Greg Lake 42 guitar
```

```
Carl Palmer 35 drums
```

```
>>>
```

```
340
```

Chapter 11 DATA STRUCTURES

Summary

This chapter introduced a number of data structures that can be used in Python. It started by showing a tuple—a list that cannot change. Then I gave examples of

the uses of a list of lists, a construct that can be used to represent grids, spreadsheets, the world of an adventure game, and anything that you can think of that is arranged as a number of rows and columns. We went through the process of taking data that was exported as a comma-separated value file (.csv) and bringing that data into a Python program as a list of lists.

I introduced the Python dictionary. In a dictionary, the data is structured as key/value pairs. You only access the data using keys. Using the `in` operator is an easy way to determine whether a key is in a dictionary. I then gave you a challenge to build a dictionary and write some code to access data found in it.

As another example, we built a dictionary of programming terms, where the keys were programming terms and each related value was the definition of the term. This demonstrated the ability to add keys and values to a dictionary. I then showed you how to iterate through a dictionary using a `for` loop. Next, I explained how lists and dictionaries could be combined to build highly complex data structures.

Finally, we went through examples of how APIs often return data using either JavaScript Object Notation (JSON) or eXtensible Markup Language (XML). There were examples of how to convert API responses into these two data structures and how to retrieve data from each.

341

CHAPTER 12

Where to Go from Here

As I said in Chapter [1](#), this book is not intended to be comprehensive. Instead, the goal is to provide you with a general understanding of programming using the Python language.

The good news is that if you have made it this far, you should have a solid understanding of most of the syntax and constructs of Python. However, the more exciting news (if you want to look at it that way) is that there is much more to explore.

This chapter discusses the following topics:

- Python language documentation
- Python Standard Library
- Python external packages
- Python development environments
- Places to find answers to questions
- Projects and practice, practice, practice

Python Language Documentation

The Python Software Foundation is the owner/developer of the Python language. In addition to the language itself, it provides a number of pages of documentation about the language, libraries, and other information useful to Python developers. The official top-level documentation for the Python language can be found at <https://docs.python.org/3.6/> (the current version at the time of writing).

343

© Irv Kalb 2018

I. Kalb, *Learn to Program with Python 3*, https://doi.org/10.1007/978-1-4842-3879-0_12

ChapTeR 12 WheRe TO GO FROM heRe

Python Standard Library

The Python Standard Library (which is installed when you download and install Python) contains a large number of built-in packages, each with many built-in functions just waiting for you to find and take advantage of them. We have only had space to scratch the surface of a few of these packages. We only talked about one or two functions of a handful of packages. Each of these packages provides functionality that is much more extensive.

There are many other built-in packages that we didn't even get to. In [Table 12-1](#), I present the ones mentioned, along with a few more of the more well-known packages available in the Python Standard Library that you may be interested in learning more about.

Table 12-1. Built-in Python Packages

Package Name	General Functionality
csv	Reading and writing comma-separated values
datetime	Basic date and time types
itertools	Functions for creating iterators for efficient looping
time	Time access and conversions
json	Creating and processing JSON-formatted data
logging	Logging facility
os	Many operating systems functions
math	Trig functions (sin, cos, tan, etc.) and constants (such as pi)

ing functions (sin, cos, tan, etc.) and constants (such as pi)

random

Generating random numbers

re

Regular expression operations

TKInter

Graphical user interface package

turtle

Turtle graphics

urllib

Opening arbitrary resources by URL

xml

Creating and processing XML-formatted data

A description of the entire Python Standard Library is at <https://docs.python.org/3.6/library/>.

An alphabetical module index is at <https://docs.python.org/3.6/py-modindex.html>.

344

ChapTeR 12 WheRe TO GO FROM heRe

Python External Packages

The Python “ecosystem” is extremely large and healthy and continues to grow. A huge number of external packages is available to Python programmers. Table [12-2](#) contains some of the most well-known external packages.

Table 12-2. External Python Packages

Package Name

General Functionality

beautifulsoup

Library for parsing hTML and XML files

django

high-level framework for building python-based web applications

flask

Microframework for building python-based web applications

Matplotlib

2D plotting library, produces publication quality figures

MySQL-Python

python connector to a MySQL database

NumPy

adds support for large, multidimensional array and high-level math functions

SciPy

Library used by scientists, analysts, and engineers doing scientific computing

Pandas

Data structures and data analysis tools

PyGame

Designed for writing games, adds support for windows, mice, and more

Requests

Makes hTTp requests in a syntax easier for humans

Scikit-learn

Software machine learning library

The official Python wiki (<https://wiki.python.org/moin/UsefulModules>) has a listing of what it thinks are the most useful external modules.

There is a site called PyPI (the Python Package Index) that calls itself “a repository of software for the Python programming language.” It may be a little difficult and intimidating to find what you are looking for there because there are over 80,000

packages cataloged. Before considering writing a package of your own, it may be worth your time checking to see whether someone has already built and published a similar module. PyPI is at <https://pypi.python.org/pypi>.

345

ChapTeR 12 WheRe TO GO FROM heRe

Python Development Environments

This book has demonstrated the use of the IDLE development environment that comes free with Python. It is a very good place to start. However, if you want to do “real” software development, you soon find that IDLE has a number of deficiencies. As mentioned earlier, the lack of line numbers is truly annoying. Perhaps most importantly, IDLE does not have a usable debugger. A *debugger* is a tool that allows a programmer to set places in a program (called *breakpoints*) where the program will stop and allow the programmer to see the value of variables, and allows the program to be executed a line at a time. IDLE claims to have a debugger, but its debugger is impossible for the average human to use. If you decide to get into serious Python development, you will probably want to graduate from using IDLE. There are a number of alternatives. I’ll tell you about some of the most popular.

Surprisingly (to me), many people develop Python code by using any basic text

editor (for example, Notepad++, TextEdit, and so on). Many programmers use a text editor like Sublime Text that (using a settings file) can be configured to have some Python-specific settings. Files are edited in the text editor and then run from the command line.

There is a well-known language-independent software development environment called Eclipse that has a plug-in called PyDev that enables Eclipse to be used as a Python development environment. If you have experience setting up an Eclipse environment, this might be a good choice for you.

The IPython Notebook is now known as the Jupyter Notebook. It is an interactive

computational environment in which you can combine code execution, rich text, mathematics, plots, and rich media. A key thing about Jupyter Notebook files is that they can contain any number of small to medium-sized pieces of Python code in a single file, and you can run any of them separately. It is an excellent environment for demonstrations and classroom use and for sharing code. It allows a wide variety of documentation types, including text with special fonts, images, YouTube videos, and so on.

PyCharm (by JetBrains) is a serious, full-featured Python IDE (short for *integrated development environment*). It comes in two flavors: Community Edition (free) and Professional Edition (paid). Beginning programmers should find the free Community Edition to have everything you need. The Professional Edition has additional features that make it worthwhile to someone who is developing Python code for a living.

Yet another option is Visual Studio from Microsoft. Visual Studio is a generic development environment that markets itself as “any language, any OS.” It has full support for Python with a downloadable source plug-in.

346

ChapTeR 12 WheRe TO GO FROM heRe

Places to Find Answers to Questions

The official Python documentation is a great place to go for detailed information on any Python syntax or documentation on any Python Standard library call.

Programmers often go to the web site www.stackoverflow.com to ask and answer programming questions. If you are stuck trying to figure out how to code something in Python, try going there and searching through the questions and answers. Often, you find that someone else has had the same question before you and other programmers have chimed in with answers. (When you start to feel comfortable with the language, try to answer some questions posed there.) Many major cities have a local “user group” where programmers get together for

talks and/or socialization. Python user groups are sometimes called PIGgies, for Python Interest Groups. A listing of many of these groups is at <https://wiki.python.org/moin/>

[LocalUserGroups.](#)

Although there are many local conferences about Python, PyCon (Python

Conference) calls itself “the largest annual gathering for the community using and developing the open-source Python programming language.” Go to www.pycon.org for details.

Projects and Practice, Practice, Practice

The only way to truly learn programming is through practice. As with a foreign language, it takes time to feel comfortable using a computer programming language. With experience, you start to recognize useful patterns in programming problems and in your solutions. To that end, I suggest that you take the time to work on developing projects on your own to gain experience. The following are some suggestions for projects that you should be able to build, using just the information presented in this book. These are all text-based projects:

- *Rock, paper, scissors*: The user chooses rock, paper, or scissors by

entering the first letter, and the computer randomly chooses one.

Rock crushes scissors, paper covers rock, scissors cuts paper.

- *Hangman*: Challenge the user to discover a randomly chosen word

within a given number of guesses of individual letters

within a given number of guesses of individual letters.

347

ChapTeR 12 WheRe TO GO FROM heRe

- *Blackjack*: Build a game of 21, where the player plays against the dealer. Add in a betting system and keep track of how much the player wins or loses.
- *Craps*: The rules are a little complicated, but this is a good programming challenge. A betting system for wins and losses makes this fun.
- *Flash cards*: Build a generic flash card testing program. Build a program that reads a file of questions and answers. Read in the file, randomize the questions, pose them to the user, and compare their answers to correct answers. For an extra challenge, allow the program to handle multiple forms of correct answers.
- *Adventure game*: Take the program we built that demonstrated the concept of a list of lists to represent a grid and turn it into a real game. Add battles against monsters, treasure hunts, and anything else you like.
- *Use an API*: Like our stock price information and weather information examples, find a publicly available API that gives you data about some topic that you care about. Build a user interface that allows the user to get the information they are interested in.

allows the user to get the information they are interested in.

Summary

This chapter was more of a reference chapter, suggesting places to go to get more information about Python. I provided links to the official documentation, where to find more information about the Standard Python Library, and external packages. I gave a listing of what I consider some of the important packages within the Standard Python Library and some of the most important external packages.

I discussed a number of alternative development environments that you can use when you outgrow IDLE. Then I listed a number of sites, groups, and conferences

that you can contact for more information about what's going on with the language.

I wrapped up with suggestions for projects that you might consider building to give you experience.

Most importantly, becoming a good Python programmer requires practice, practice, practice!

348

Index

A

Building blocks of programming

coding, [14](#)

Absolute path, [247](#)

data

Absolute value program, [120–123](#)

Booleans, [16](#)

Adding game

definition, [14–15](#)

add loop and score counter, [261–263](#)

floating-point numbers, [15](#)

join function, [266](#)

integer numbers, [15](#)

random numbers, [260–261](#)

strings, [16](#)

[split function, 267](#)

Built-in functions

user name, [268](#)

arguments, [52](#)

version 3, [263–264](#)

assignment statement, [53–54](#)

Adventure games, [307–310](#)

concatenation, [61–63](#)

Analogy for building software

dynamically typed

recipe example

language, [55](#)

baking chocolate cake, [68–69](#)

function call, [52](#)

crack eggs into bowl, [69–71](#)

overview, [51–52](#)

Assignment operator, [25](#)

real programs, [58–60](#)

Assignment statements, [24–26, 32–33](#)

type function, [53–55](#)

user input function, [55–56](#)

B

Built-in list operations, [214–215](#)

Blackjack game, [110](#)

Bytecode, [281](#)

[blend function, 76–78](#)

Boolean expression, [16, 18, 108](#)

conditional logic

C

and operator, [133–134](#)

Camelcase naming

not operator, [132–133](#)

convention, [29](#)

or operator, [135](#)

Chevron prompt, [4](#)

elif statement, [117](#)

C language, [31](#)

else statement, [111–112](#)

Class scope, [96](#)

if statement, [111](#), [136](#)

Coin flipping, [157–158](#)

349

© Irv Kalb 2018

I. Kalb, *Learn to Program with Python 3*, <https://doi.org/10.1007/978-1-4842-3879-0>

Index

Comma-separated value (.csv) file

programming terms, [322–323](#)

grades spreadsheet, [311–312](#)

U.S. states population, [320–321](#),

if/elif/else statements, [313–314](#)

[323–324](#)

letterGrade function, [315](#)

grid/spreadsheet, [306](#)

Comments

JSON, [328](#)

after a line of code, [44](#)

lists of lists, [305](#)

full-line, [43](#)

OpenWeatherMap API, [331–333](#)

multiline, [44–45](#)

tuples, [302–304](#)

Comparison operator, [109](#)

weather data program, [331–333](#)

Compile error, *see* Syntax error

XML, [334–335](#), [337](#)

Concatenation operator, [193–194](#)

Debugger tool, [346](#)

Conditional logic

Dice program, [210–211](#), [213](#)

and operator, [133–134](#)

not operator, [132–133](#)

E

or operator, [135](#)

Conversion functions

Elements, [185–186](#)

float function, [57](#)

elif statement, [115, 117–119](#)

int [function, 57](#)

else statement, [111–115](#)

[str function, 57–58](#)

Empty list, [187](#)

Errors

D

exception, [48](#)

logic, [49](#)

Data

syntax, [46–47](#)

Booleans, [16](#)

Exception error, [48](#)

collections of, [184](#)

[definition, 14–15](#)

floating-point numbers, [15](#)

F

integer numbers, [15](#)

File input/output

strings, [16](#)

data saving and reading, [257, 259](#)

Data structure

file handle, [250–251](#)

adventure games, [307](#)

FileReadWrite.py, [271–272](#)

.csv file

import modules, [255–257](#)

grades spreadsheet, [311–312](#)

MultiLineData.txt, [273](#)

if/elif/else statements, [313–314](#)

os module, [251–252](#)

letterGrade function, [315](#)

path to text file, [247, 249](#)

dictionary

reading from and writing to file, [249](#)

houseDict, [317–320](#)

reusable functions, [252–253](#)

key/value pairs, [316](#)

save file, [246](#)

and list, [325–326, 328](#)

writing to and reading from file, [254](#)

350

Index

Five-dollar bills program, [39–40](#)

if statement

Floating-point numbers, [15–17](#)

blackjack game, [110](#)

Flowchart

Boolean expression, [108](#)

[definition, 104](#)

comparison operator, [109](#)

if/else statement, [112–114](#)

gas station, [111](#)

if statement (*see* if statement)

syntax, [108](#)

for store checkout, [105](#)

Infinite loop, [148, 160](#)

for weekday morning routine, [107](#)

Integer numbers, [15, 17, 36](#)

Full-line comment, [43](#)

Inventory game, [216](#)

Function calls, [63–64](#)

IPython Notebook, *see* Jupyter Notebook

isEven function, [128–130](#)

G

isRectangle function, [130–131](#)

isSquare function, [125–126, 128](#)

General-purpose programming

language, [2](#)

getGroceries function

J

arguments, [76](#)

JavaScript Object Notation (JSON), [328](#)

def statement, [75–76](#)

and XML format, [338–340](#)

hard-coding, [76](#)

[join function, 266–267](#)

main code, [74](#)

Jupyter Notebook, [346](#)

parameters, [78–81](#)

runtime error, [98–99, 101](#)

K

Grading program, [120](#)

Keywords, [29–30](#)

H

L

Hard-coding, [76](#)

Hello World program, [4–5](#)

len function, [196–198](#)

Lists, [185](#)

Logic error, [49](#)

I

Loops

IDLE

break statement, [161–162, 175](#)

dedent/outdent, [72](#)

coding challenge, [179–181](#)

install, [3](#)

coin flipping, [157–158](#)

on multiple platforms, [8](#)

continue statement, [175–176](#)

open file, [7, 72](#)

empty string, [163](#)

quit buttons, [7](#)

error-checking utility

if/else statements, [114](#)

functions, [178–179](#)

351

Index

Loops (*cont.*)

one-dollar bills and five-dollar

flowchart, [146](#)

bills, [39–40, 42](#)

Guess the Number

ten-dollar bills and twenty-dollar

program, [164, 166–170](#)

bills, [43](#)

increment and decrement, [151–152](#)

Multiline comment, [44](#)

infinite, [148, 160](#)

Multiple choice test, [275](#)

playing multiple games, [171–173](#)

pseudocode, [164](#)

N

Python's built-in packages, [154–155](#)

random number generation, [155–157](#)

Naming convention, [28–29](#)

real-life example, [146–147](#)

Negative index, [192–193](#)

rock-paper-scissors, [158–159](#)

negativePositiveZero function, [123–124](#)

running a program, [152–154](#)

Nested if statement, [111](#)

target number, [150](#)

Numbers list

try/except block, error

calculate total, [206–207](#)

detection, [173–174](#), [176–178](#)

integer, [208–209](#)

user's view of game, [144–145](#)

range function, [207–208](#)

while statement, [147–148](#)

while True, [160](#)

O

One-dollar bills program, [39–40](#)

M

One Infinite Loop Drive, [149](#)

Mad Libs game

OpenWeatherMap API, [331–333](#)

choice function, [202](#)

OpenWeatherMap.org, [294–295, 297](#)

chooseRandomFromList

Order of operations, [38–39](#)

function, [201–202](#)

concatenation operator, [193–194](#)

P, Q

[len function, 197](#)

pool of names, [195](#)

Parameter, [77](#)

random package, [195–196](#)

Parameter variable, [77](#)

sentences, [198, 200](#)

Parentheses Exponents Multiplication

Magic 8-Ball program, [12, 14](#)

Division Addition Subtraction

Math game, [112](#)

(PEMDAS), [38–39](#)

Math operators, [35–37, 136](#)

Persistent data, [245](#)

Mental model, [21](#)

PIGgies, [347](#)

Methods of an object, [214](#)

Pizza toppings program, [217](#)

Modulo operator, [37](#)

Print statements, [33–35](#)

MoneyInWallet.py program

Prompt, [4](#)

352

Index

Pseudocode, [164](#)

S

PyCharm (by JetBrains), [346](#)

Screen scraping approach, [287](#)

PyDev [plug-in, 346](#)

Scripting language, [2](#)

Pythagorean theorem, [40–41](#)

Shipping program, [137–138, 140](#)

Python

Shopping list

advantage, [4](#)

assignment statement, [189](#)

constant, [92](#)

change value, [192](#)

external packages, [345](#)

elements, [185](#)

functions, [89](#)

enter an integer, [190–191](#)

installing, [2–3](#)

for loop, [205–206](#)

language documentation, [343](#)

for statement, [204](#)

Mac, [8](#)

index, [188](#)

open source, [2](#)

[len function, 197](#)

print statements, [6](#)

myList bracket 2 bracket, [190](#)

.py extension, [7](#)

negative index, [192–193](#)

[run](#), [7](#)

print statement, [189](#)

save, [6–7](#)

square bracket syntax, [186](#)

Shell, [3–4](#)

strings, [185](#)

Python Conference (PyCon), [347](#)

while statement, [203](#)

Python Package Index (PyPI), [345](#)

Shorthand naming convention, [41–43](#)

Python Standard Library, [154](#), [344](#)

split function, [267](#)

Stepwise refinement, [69](#)

R

Stock price

Random-access memory (RAM), [19](#),

APIs

[21–22](#)

key, [289](#), [291](#)

Random numbers program, [13–14](#)

program code, [292–293](#)

Relative paths, [247](#)

for retrieving data, [288](#)

Request/response model, [283–285](#)

string, [291](#)

return statement

Apple stock, [285–287](#)

addTwo [function](#), [83](#)

HTML text, [286](#)

calculateAverage [function](#), [83](#)

name/value pair, [289](#)

generic form, [83](#)

screen scraping, [287](#)

return many values, [85–86](#)

stock symbol, [285](#)

returns no value, [84](#)

URL, [288–289](#)

Reusable file I/O functions, [252–253](#)

Strings, [16–17](#)

Right triangle program, [90–91](#)

accessing characters, [227–228](#)

Runtime error, *see* Exception error

countCharInString, [237](#)

353

Index

Strings (*cont.*)

getGroceries function (*see* getGroceries

directory style, [241–242](#)

function)

findChar, [237–238](#)

global variable, [93–94](#), [97–98](#)

for loop, [229–230](#)

local variable, [95–98](#)

indexing characters, [226–227](#)

startingValue, [81](#)

[len function, 226](#)

not changeable, [236](#)

V

operations, [238–241](#)

slicing

Variables

bread, [230–231](#)

assignment statement, [24–26](#)

list, [236](#)

case sensitive, [31](#)

month number, [232–234](#)

global, [93–94](#), [97–98](#)

syntax, [231](#), [235](#)

local, [95–98](#)

targetString, [237–238](#)

names

while loop, [228](#)

in calls and functions, [86–87](#)

Sublime Text editor, [346](#)

conventions, [28–29](#)

Syntax error, [46–47](#)

illegal, [28](#)

Python's rules, [27](#)

RAM, [19](#), [21–22](#)

T

visualize envelope, [22–24](#)

Temperature conversion functions, [88–89](#)

Traceback, [46](#), [98–101](#)

W

Tuples, [302–304](#)

Weather data program, [331–333](#)

Weather information program, [294–297](#)

U

Whitespace, [45](#)

Universal Resource Locator (URL),

WidgetsRUs.com, [18](#)

[297–299](#)

Working wage program, [40](#)

User-defined functions

Wrong answer, *see* Logic error

blend function, [77](#)

calculates average of

X, Y, Z

numbers, [81, 86–87](#)

[definition, 72](#)

XML data, [334–338](#)

Document Outline

- [Table of Contents](#)
- [About the Author](#)
- [About the Technical Reviewer](#)
- [Acknowledgments](#)
- [Chapter 1: Getting Started](#)
 - [What Is Python?](#)
 - [Installing Python](#)
 - [IDLE and the Python Shell](#)
 - [Hello World](#)
 - [Creating, Saving, and Running a Python File](#)
 - [IDLE on Multiple Platforms](#)
 - [Summary](#)
- [Chapter 2: Variables and Assignment Statements](#)
 - [A Sample Python Program](#)
 - [The Building Blocks of Programming](#)
 - [Four Types of Data](#)
 - [Integers](#)
 - [Floats](#)
 - [Strings](#)
 - [Booleans](#)
 - [Examples of Data](#)
 - [Form with Underlying Data](#)
 - [Variables](#)
 - [Assignment Statements](#)
 - [Variable Names](#)
 - [Naming Convention](#)
 - [Keywords](#)
 - [Case Sensitivity](#)
 - [More Complicated Assignment Statements](#)
 - [Print Statements](#)
 - [Simple Math](#)
 - [Order of Operations](#)
 - [First Python Programs](#)
 - [Shorthand Naming Convention](#)

- [Adding Comments](#)
 - [Full-Line Comment](#)
 - [Add a Comment After a Line of Code](#)
 - [Multiline Comment](#)
- [Whitespace](#)
- [Errors](#)
 - [Syntax Error](#)
 - [Exception Error](#)
 - [Logic Error](#)
- [Summary](#)
- [Chapter 3: Built-in Functions](#)
 - [Overview of Built-in Functions](#)
 - [Function Call](#)
 - [Arguments](#)
 - [Results](#)
 - [Built-in type Function](#)
 - [Getting Input from the User](#)
 - [Conversion Functions](#)
 - [int Function](#)
 - [float Function](#)
 - [str Function](#)
 - [First Real Programs](#)
 - [Concatenation](#)
 - [Another Programming Exercise](#)
 - [Using Function Calls Inside Assignment Statements](#)
 - [Summary](#)
- [Chapter 4: User-Defined Functions](#)
 - [A Recipe as an Analogy for Building Software](#)
 - [Ingredients](#)
 - [Directions](#)
 - [Definition of a Function](#)
 - [Building Our First Function](#)
 - [Calling a User-Defined Function](#)
 - [Receiving Data in a User-Defined Function: Parameters](#)
 - [Building User-Defined Functions with Parameters](#)
 - [Building a Simple Function That Does Addition](#)
 - [Building a Function to Calculate an Average](#)
 - [Returning a Value from a Function: The return Statement](#)
 - [Returning No Value: None](#)

- [Returning More Than One Value](#)
- [Specific and General Variable Names in Calls and Functions](#)
- [Temperature Conversion Functions](#)
- [Placement of Functions in a Python File](#)
- [Never Write Multiple Copies of the Same Code](#)
- [Constants](#)
- [Scope](#)
- [Global Variables and Local Variables with the Same Names](#)
- [Finding Errors in Functions: Traceback](#)
- [Summary](#)
- [Chapter 5: if, else, and elif Statements](#)
 - [Flowcharting](#)
 - [The if Statement](#)
 - [Comparison Operators](#)
 - [Examples of if Statements](#)
 - [Nested if Statement](#)
 - [The else Statement](#)
 - [Using if/else Inside a Function](#)
 - [The elif Statement](#)
 - [Using Many elif Statements](#)
 - [A Grading Program](#)
 - [A Small Sample Program: Absolute Value](#)
 - [Programming Challenges](#)
 - [Negative, Positive, Zero](#)
 - [isSquare](#)
 - [isEven](#)
 - [isRectangle](#)
 - [Conditional Logic](#)
 - [The Logical not Operator](#)
 - [The Logical and Operator](#)
 - [The Logical or Operator](#)
 - [Precedence of Comparison and Logical Operators](#)
 - [Booleans in if Statements](#)
 - [Program to Calculate Shipping](#)
 - [Summary](#)
- [Chapter 6: Loops](#)
 - [User's View of the Game](#)
 - [Loops](#)
 - [The while Statement](#)

- [First Loop in a Real Program](#)
- [Increment and Decrement](#)
- [Running a Program Multiple Times](#)
- [Python's Built-in Packages](#)
- [Generating a Random Number](#)
- [Simulation of Flipping a Coin](#)
- [Other Examples of Using Random Numbers](#)
- [Creating an Infinite Loop](#)
- [A New Style of Building a Loop: while True, and break](#)
- [Asking If the User Wants to Repeat: the Empty String](#)
- [Pseudocode](#)
- [Building the Guess the Number Program](#)
- [Playing a Game Multiple Times](#)
- [Error Detection with try/except](#)
- [The continue Statement](#)
- [Full Game](#)
- [Building Error-Checking Utility Functions](#)
- [Coding Challenge](#)
- [Summary](#)
- [Chapter 7: Lists](#)
 - [Collections of Data](#)
 - [Lists](#)
 - [Elements](#)
 - [Python Syntax for a List](#)
 - [Empty List](#)
 - [Position of an Element in a List: Index](#)
 - [Accessing an Element in a List](#)
 - [Using a Variable or Expression as an Index in a List](#)
 - [Changing a Value in a List](#)
 - [Using Negative Indices](#)
 - [Building a Simple Mad Libs Game](#)
 - [Adding a List to Our Mad Libs Game](#)
 - [Determining the Number of Elements in a List: The len Function](#)
 - [Programming Challenge 1](#)
 - [Using a List Argument with a Function](#)
 - [Accessing All Elements of a List: Iteration](#)
 - [for Statements and for Loops](#)
 - [Programming Challenge 2](#)
 - [Generating a Range of Numbers](#)

- [Programming Challenge 3](#)
- [Scientific Simulations](#)
- [List Manipulation](#)
- [List Manipulation Example: an Inventory Example](#)
- [Pizza Toppings Example](#)
- [Summary](#)
- [Chapter 8: Strings](#)
 - [len Function Applied to Strings](#)
 - [Indexing Characters in a String](#)
 - [Accessing Characters in a String](#)
 - [Iterating Through Characters in a String](#)
 - [Creating a Substring: A Slice](#)
 - [Programming Challenge 1: Creating a Slice](#)
 - [Additional Slicing Syntax](#)
 - [Slicing as Applied to a List](#)
 - [Strings Are Not Changeable](#)
 - [Programming Challenge 2: Searching a String](#)
 - [Built-in String Operations](#)
 - [Examples of String Operations](#)
 - [Programming Challenge 3: Directory Style](#)
 - [Summary](#)
- [Chapter 9: File Input/Output](#)
 - [Saving Files on a Computer](#)
 - [Defining a Path to a File](#)
 - [Reading from and Writing to a File](#)
 - [File Handle](#)
 - [The Python os Package](#)
 - [Building Reusable File I/O Functions](#)
 - [Example Using Our File I/O Functions](#)
 - [Importing Our Own Modules](#)
 - [Saving Data to a File and Reading It Back](#)
 - [Building an Adding Game](#)
 - [Programming Challenge 1](#)
 - [Programming Challenge 2](#)
 - [Writing/Reading One Piece of Data to and from a File](#)
 - [Writing/Reading Multiple Pieces of Data to and from a File](#)
 - [The join Function](#)
 - [The split Function](#)
 - [Final Version of the Adding Game](#)

- [Writing and Reading a Line at a Time with a File](#)
- [Example: Multiple Choice Test](#)
- [A Compiled Version of a Module](#)
- [Summary](#)
- [Chapter 10: Internet Data](#)
 - [Request/Response Model](#)
 - [Getting a Stock Price](#)
 - [Pretending to Be a Browser](#)
 - [API](#)
 - [Requests with Values](#)
 - [API Key](#)
 - [Example Program to Get Stock Price Information Using an API](#)
 - [Example Program to Get Weather Information](#)
 - [URL Encoding](#)
 - [Summary](#)
- [Chapter 11: Data Structures](#)
 - [Tuples](#)
 - [Lists of Lists](#)
 - [Representing a Grid or a Spreadsheet](#)
 - [Representing the World of an Adventure Game](#)
 - [Reading a Comma-Separated Value \(.csv\) File](#)
 - [Dictionary](#)
 - [Using the in Operator on a Dictionary](#)
 - [Programming Challenge](#)
 - [A Python Dictionary to Represent a Programming Dictionary](#)
 - [Iterating Through a Dictionary](#)
 - [Combining Lists and Dictionaries](#)
 - [JSON: JavaScript Object Notation](#)
 - [Example Program to Get Weather Data](#)
 - [XML Data](#)
 - [Accessing Repeating Groupings in JSON and XML](#)
 - [Summary](#)
- [Chapter 12: Where to Go from Here](#)
 - [Python Language Documentation](#)
 - [Python Standard Library](#)
 - [Python External Packages](#)
 - [Python Development Environments](#)
 - [Places to Find Answers to Questions](#)
 - [Projects and Practice, Practice, Practice](#)

- [Summary](#)
- [Index](#)