The Biggest Python Programming eBook of 2019

Python® with Raspberry Pi®

Python® with Raspberry Pi®

Alex Bradbury & Ben Everard

This edition first published 2014 © 2014 Alex Bradbury and Ben Everard
Registered office **John Wiley & Sons Ltd,** The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.
The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The publisher is not associated with any product or vendor mentioned in this book. This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and/or other countries, and may not be used without written permission. Python is a

registered trademark of the PSF (Python Software Foundation). Raspberry Pi and the Raspberry Pi Logo are a registered trademark of the Raspberry Pi Foundation, which is a UK registered charity. Minecraft is a trademark of Mohang. Mac OS, iPad, and iPhone are registered trademarks of Apple Inc. Pi Cobbler is a trademark of Adafruit. All other trademarks are the property of their respective owners. John Wiley & Sons, Ltd. is not associated with any product or vendor mentioned in the book.

A catalogue record for this book is available from the British Library.
ISBN 978-1-118-71705-9 (paperback); ISBN 978-1-118-71703-5 (ePub); 978-1-118-71702-8 (ePDF) Set in 10 pt and ChaparralPro-Light by TCS/SPS
Printed simultaneously in the United States and the United Kingdom

To Kat for her continuing support, Mum and Dad for encouraging me to learn to program on the Commodore 64, Zappa for coping with continual disruption, and every single free and open source software developer for being awesome.

—Ben

Publisher’s Acknowledgements

Some of the people who helped bring this book to market include the following:
Editorial and Production Marketing

VP Consumer and Technology Publishing Director:
Michelle Leete

Marketing Manager: Lorna Mein

Associate Director–Book Content Management:
Martin Tribe

Associate Publisher: Chris Webb
Executive Commissioning Editor: Craig Smith
Project Editor: Kezia Endsley
Copy Editor: Kezia Endsley
Technical Editor: Paul Hallett
Editorial Manager: Jodi Jensen
Senior Project Editor: Sara Shlaer

Proofreader: Linda Seifert
Editorial Assistant: Annie Sullivan
Marketing Assistant: Polly Thomas

About the Authors

BEN EVERARD is a Linux geek with a penchant for writing. He's a founder and director of Linux Voice (http://linuxvoice.com), and his musings can be found on the pages of their magazine and in their podcast. Previously, he's worked as a technical editor at Linux Format, and as a country manager for NoPC, where he oversaw the testing and deployment of computers to schools in Tanzania. Once upon a time, he was an IT consultant, but that was so long ago he can't remember it.

He's moved house and country so many times in the past six years, he's practically nomadic, although these days he can usually be found in the West Country, England. This is his first book.

ALEX BRADBURY is a compiler, hacker, Linux geek, and Free Software enthusiast. His involvement with the Raspberry Pi started when the first alpha boards were produced. He quickly got sucked in, leading Linux software development efforts for the platform. Still a steady contributor, he's currently focusing on finishing his PhD at the University of Cambridge Computer Laboratory on compilation techniques for a novel many-core architecture. He's on Twitter as @asbradbury, or else you can email him at asb@asbradbury.org.

Acknowledgments

Many people have helped make this book possible. At Wiley, Kezia Endsley and Craig Smith saw the book through from its inception. Thank you also to Erin Zeltner for making the words look fantastic and making sure they fit on the pages properly.

There are so many more people that also deserve a huge thank you. There couldn't be a programming book without a programming environment. Python on the Raspberry Pi is the work of literally thousands of programmers, many of them unpaid. They all deserve acknowledgment, but because of space, we'll only mention three—Guido van Rossum, Linux Torvalds, and Richard Stallman.

Of course, the software needs hardware to run on, so we'd also like to extend thanks to Eben Upton and the entire Raspberry Pi Foundation.
Any and all mistakes are, of course, the sole responsibility of the authors.

Contents

Introduction 1
What Is Programming? 1
Why the Raspberry Pi? 2
How Does this Book Work? 2

CHAPTER 1
Getting Up and Running 5

Setting Up Your Raspberry Pi 5
Solving Problems 6
A Quick Tour of Raspbian 7
Using LXDE (Lightweight X11 Desktop Environment) 7
Using the Terminal 8
Changing Configurations with Raspi-Config 10
Installing Software 10
Python 3 11
The Python Interpreter 11
Running Python Programs 11
Summary 12

CHAPTER 2
A Really Quick Introduction to Python 13

Drawing Picture with Turtles 13
Using Loops 16
Conditionals: if, elif, and else 18
Using Functions and Methods to Structure Code 20
A Python Game of Cat and Mouse 21
Understanding Variables 24
Defining Functions 24
Looping Through the Game 25
Summary 26

CHAPTER 3
Python Basics 27

Variables, Values, and Types 27
Values Have Types 28
Storing Numbers 29

Keeping Text in Strings 30
Boolean: True or False 31
Converting Between Data Types . . . . . .

. 32
Test Your Knowledge 32
Exercise 1 33
Storing Values in Structures 33
Non-Sequential Values in Dictionaries and Sets 37
Test Your Knowledge 38
Controlling the Way the Program Flows 38
Moving Through Data with for Loops 39
Going Deeper with Nested Loops 40
Branching Execution with if Statements 41
Catching Exceptions 42
Exercise 2 43
Making Code Reusable with Functions 43
Optional Parameters 46
Bringing Everything Together 46
Exercise 3 48
Building Objects with Classes 49
Getting Extra Features from Modules 54
Summary 55
Solutions to Exercises 56

Exercise 1 56
Exercise 2 56
Exercise 3 56

CHAPTER 4
Graphical Programming 57

Graphical User Interface (GUI) Programming 58
Adding Controls 60
Test Your Knowledge 62
Exercise 1 62
Creating a Web Browser 62
Adding Window Menus 71
Test Your Knowledge 73
Exercise 2 73

Summary 74
Solutions to Exercises 74
Exercise 1 74
Exercise 2 76

CHAPTER 5

Creating Games 77

Building a Game 79
Initialising PyGame 82
Creating a World 86
Detecting Collisions 88
Moving Left and Right 90
Reaching the Goal 92
Making a Challenge 93
Making It Your Own 97
Adding Sound 98
Adding Scenery 99
Adding the Finishing Touches 101
Taking the Game to the Next Level 102
Realistic Game Physics 103
Summary 108

CHAPTER 6
Creating Graphics with OpenGL 109

Getting Modules 110
Creating a Spinning Cube 110
Vectors and Matrices . . . . . .

. . . . 112
Bringing It All Together 116
Let There Be Light 120
Calculating the Distance of the Point from the Light Source 120
Calculating Reflecting Angles 121
Making the Screen Dance 126
Building the 3D Model 128
Calculating the Sound Level 129
Taking Things Further 135
Adding Some Texture 136
Summary 136

CHAPTER 7
Networked Python 139

Understanding Hosts, Ports, and Sockets 139
Locating Computers with IP Addresses 140
Building a Chat Server 141
Tweeting to the World 144
Weather Forecasts with JSON 147
Testing Your Knowledge 149
Exercise 1 149

Getting On the Web 149
Making Your Website Dynamic 151
Using Templates 153
Sending Data Back with Forms 153
Exercise 2 155
Keeping Things Secure 155
Summary 159
Solutions to Exercises 160
Exercise 1 160
Exercise 2 161

CHAPTER 8
Minecraft 163

Exploring Minecraft 164
Controlling Your Minecraft World 164
Creating Minecraft Worlds in Python 165
Building Worlds 166
Drawing Pictures 166
Taking Things Further 169
Making the Game Snake 169
Moving the Snake . . . . . .

. 173
Growing the Snake 173
Adding the Apples 174
Taking Things Further 175
Summary 175

CHAPTER 9
Multimedia 177

Using PyAudio to Get Sound into Your Computer 177
Recording the Sound 179
Speaking to Your Pi 180

Asking the Program Questions 181
Putting It All Together 182
Taking Things Further 184
Making Movies 184
Using USB Webcams 185
Adding Computer Vision Features with OpenCV 187
Taking Things Further 190
Using the Raspberry Pi Camera Module 190
Creating Live Streams . . . . . .

. 193
Taking Things Further 196
Summary 196

CHAPTER 10
Scripting 197

Getting Started with the Linux Command Line 197
Using the Subprocess Module 200
Command-Line Flags 202
Regular Expressions 203
Testing Your Knowledge 206
Scripting with Networking 207
Bringing It All Together 209
Working with Files in Python 216
Summary 217

CHAPTER 11
Interfacing with Hardware. 219

Setting Up Your Hardware Options 219
Female to Male Jumper Wires 219
Pi Cobbler 220
Solderless Breadboard 221

Stripboards and Prototyping Boards 221
PCB Manufacturing 222
Getting the Best Tools 222
Wire Cutters/Strippers 222
Multimeters 222
Soldering Irons 222
Hardware Needed for this Chapter 223
The First Circuit 224
Protecting Your Pi 228
Power Limits . . . . 229 Getting Input . . . . . 229 Expanding the GPIO Options with I2C, SPI, and Serial . . . . . 231 The SPI Communications Protocol . . . . . 231 The I2C Communications Protocol . . . . . 235 The Serial Communications Protocol . . . . . 236 Taking the Example Further . . . . . 236 Arduino . . . . . 236 PiFace . . . . . 237 Gertboard . . . . . 237 Wireless Inventor’s Kit . . . . . 238 Trying Some Popular Projects . . . . . 238 Robots . . . . . 238 Home Automation . . . . 239 Burglar Alarms . . . . . 239 Digital Art . . . . . 239 Summary 239

CHAPTER 12
Testing and Debugging 241

Investigating Bugs by Printing Out the Values 241
Finding Bugs by Testing 245
Checking Bits of Code with Unit Tests 245
Getting More Assertive 250
Using Test Suites for Regression Testing 252
Testing the Whole Package 253
Making Sure Your Software's Usable 254
How Much Should You Test? 254
Summary 255

Index 257

Introduction

COMPUTERS AREN'T JUST beige square things we use for work, they're everything that has a programmable processing unit at its heart. Games consoles, smartphones, GPS units, tablets and a mind-boggling range of other devices all work in the same way. They're all computers, and they've taken over the world. They're the things we use for work, for communications, and for relaxation. In fact, it's hard to think of an area that hasn't been taken over by computers.

Marketing people like to tell you that devices with embedded computers are smart (smartphones, smart TVs, smart watches, and so on), but the truth is they're not. The processing units are just bits of silicon that follow a set of instructions. The "smart" in a smartphone doesn't come from the computer chips, but from the people who program them.

Computers are the most powerful tools mankind has ever created, yet they're under-utilised because few people know how to unleash their full potential. In a world where everything is a computer, the most important people are the programmers who can realise their full power. Programming, then, is an essential skill that's only going to become more and more important in the future.

What Is Programming?

Computers, as we've said, aren't smart. They just follow a simple list of instructions one-by-one until they reach the end. That list of instructions is a program. Programming, then, is the process of taking a task, splitting it up into steps, and writing it down in a language the computer can understand.

The Raspberry Pi can understand many languages, but in this book, you'll learn about Python 3. It's a powerful language, and easy to learn.

This book is for people who want to learn about computer programming and who have a Raspberry Pi. You don't need any special skills or prior experience to work your way through this book, and it doesn't matter if you're not a classic geek who reads comics and watches Sci-Fi, and it doesn't matter if you are. As long as you fit those two basic criteria, this is the book is for you.

By the end of this book, you'll have a good grasp of Python 3, and you'll be familiar with many of the most useful modules (add-ons). Using these, you'll be able to control almost every aspect of your Pi. You'll make it interact with the world around through the General Purpose Inputs and Outputs (GPIOs), and communicate over the Internet. You'll give it vision so it can snap photos and know what it's looking at. You'll make games and manipulate three-dimensional worlds. In short, this is a book about how to utilise your Raspberry Pi to its fullest potential.

Why the Raspberry Pi?

There are a few things that make the Raspberry Pi a great device on which to learn programming. Firstly it's cheap. At around a tenth of the price of a low-end PC, it's cheap enough to have in addition to your main computer. This is useful because programmers tend to tinker with their development machine, and

tinkering can break things. Generally this doesn't damage the machine itself, but it can require you to reinstall the system, which can mean a bit of lost data, and it can put the machine out of action for a few hours. If you have a Pi that's used just for development, this isn't a problem; however, if your only computer is shared with a few other people, they may be a bit put out by this.

Secondly, the Pi is raw. It doesn't come hidden away in a box, or in a complete system. This means that you get to decide what sort of system you want to make. You can enclose it in a case should you wish, or you can run it naked. You have access to GPIOs that many machines don't have. Most computers come pre-packaged for a particular purpose (a tablet for surfing the web or playing games, a games console for watching movies or playing games, a laptop for working or playing games, and so on). A Raspberry Pi can turn its hand to any of these things with just a little technical know-how.

Thirdly, the Raspberry Pi runs Linux. This is an operating system a bit like Windows or Mac OS X. It provides a windowing system and a text-based interface for controlling the Pi. If you haven't used Linux before, you'll notice a few differences between it and the system you're used to. For budding programmers, though, the most important difference is that Linux is far more flexible than the alternatives. Just as the physical design of the Raspberry Pi encourages experimentation, so does the operating system.

How Does this Book Work?

Chapters 1–3 are all about getting started with Python on your Raspberry Pi. At the end of them, you'll have a pretty good idea of what Python programming is about. The rest of the book is split into chapters that deal with different uses, such as games or multimedia. These

chapters deal with different areas of Python, so generally, you don't need to have read one chapter to understand the next (there are a couple of times where we refer back to something, but we make it clear what's going on when we do).

This means that you can go through this second part of the book in whatever order you want. For example, if you have a particular interest in multimedia, you can skip ahead to that, and then come back and read the others later.

Learning to program is all about actually getting your hands dirty and programming. This means that you can't learn it by just sitting down and reading a book; you actually have to do some yourself. Throughout this book we challenge you to put what you've learned to the test. Sometimes it's through specific exercises designed to train your skills, other times it's through taking the programs we've introduced and adding your own features to them. An important part of programming is the creativity to decide what you want the program to do, so you don't have to follow our suggestions. In fact, we encourage you to treat our suggestions and code as a starting point to creating your own digital works of art.

Chapter 1
Getting Up and Running

WELCOME TO Learning Python with Raspberry Pi. In this book, you'll learn how to unlock the full power of the tiny computer, from 3D graphics to games programming to controlling electronics to tweeting. You'll see what's going on under the hood and learn how to create programs that take advantage of every feature of this minuscule computer.

Setting Up Your Raspberry Pi

To follow this book, you'll need a few bits of equipment:

- Raspberry Pi
- USB keyboard
- USB mouse
- SD card
- Monitor
- Power supply

There are also a few optional bits of kit that may help:

- Powered USB hub (highly recommended)
- Camera module
- USB webcam
- USB WiFi dongle

It is possible to do everything in this book with a model A Raspberry Pi. The real

advantage of a model B as far as programming is concerned is the network port. This port will make it easier to connect to the Internet, which you'll need to do to install some software. Any USB keyboard and mouse should work fine. Most SD cards should work, although there are a few that will cause problems. If you're unsure, buy one from a Raspberry Pi online shop (there are links to a few on http://raspberrypi.org).

The Raspberry Pi has a HDMI (high-definition multimedia interface) video output, but most monitors have VGA or DVI input. If at all possible, use a monitor that has DVI or HDMI input. A HDMI-to-DVI converter should cost only a few pounds/dollars and shouldn't detract from the image quality. HDMI-to-VGA converters are available, but they're more expensive and can cause problems, so use them only if you have no other option.

Most micro USB power supplies from reputable manufacturers should work; however, some cheap ones from no-name companies have caused problems, so if possible, don't skimp too much on this. You could use a USB cable from a normal computer to power your Pi.

Powered USB hubs are recommended for the power-related problems described later in this chapter. Not all USB hubs are powered, so make sure that whatever one you get plugs into the mains electricity to get extra power.

We talk more about camera options in Chapter 9 on multimedia. The only thing to say here is that if you do choose to get a USB webcam, make sure it's compatible with the Raspberry Pi. There's a partial list of working web cams at http://elinux.org/RPi_USB_Webcams.

You'll need to connect your Pi to the Internet to install the software you need in this book. You can do this either by plugging your Pi into your router with a network cable or by using a USB wireless dongle, which will add WiFi connectivity.

Solving Problems

The most common problems with the Raspberry Pi are power-related issues. Not all micro USB power sources can provide enough power, and it becomes more of a problem as you connect peripherals to your Pi, or when you overclock it (see Chapter 5 for more details). Powerrelated problems will usually manifest

themselves as the computer crashing, so if you find that your Pi becomes unstable, this is the best place to start. A good way to get around such issues is to connect your Pi to one power source and connect all the peripherals (keyboard, mouse, and so on) via a powered USB hub.

The second most common cause of problems with Pis is the SD card. These issues can be caused by power supply problems, or they can be problems with the cards themselves. It's important to take preventative measures here to ensure that your data is safe, and that means backups! You can use a service such as Google Drive (although this runs slowly on the Pi), or you can simply keep extra copies of any work on a USB memory stick. SD card issues will usually manifest themselves by the Pi displaying error messages when you try to start it. Most of the time you can solve the problem by reinstalling Raspbian, but if this doesn't work, you'll need to get a new SD card.
If neither of these help, then you'll need to dig a little deeper. The most useful places to look are the kernel buffer and the system log file. The kernel buffer is usually best if you're having problems with hardware, such as a USB device not working. If you open LXTerminal and type:

```
dmesg
```

It will output all the messages from the Linux Kernel. The last ones are the most recent and should show any problems.
The system log file (often called syslog) can be displayed with:

```
cat /var/log/syslog
```

Again, the most recent messages will be at the end. The information in both of these can be somewhat cryptic. If you still can't work out the problem after reading these, the best place to go is the Raspberry Pi forums at www.raspberrypi.org/phpBB3/. There's a community of helpful people who should be able to point you in the right direction.

A Quick Tour of Raspbian

This is a book about programming, not about generally using Raspbian, so we won't dwell on it too much, but you'll find it useful to know a bit about what's going on.

There are a few operating systems available for the Raspberry Pi, but the instructions in this book are all based on Raspbian, which is the default operating

system, and the best choice for a new user. If you have some experience with Linux, you could use Arch or Fedora if you like, but you'll have to change the apt-get commands to ones suitable for your package manager.

The easiest way to install Raspbian on your Pi is using NOOBS, which is available from www. raspberrypi.org/downloads. You'll also find a quick start guide at that website that will tell you everything you need to know to get up and running.

There are two different ways of interacting with Raspbian—from the terminal and using the graphical system (LXDE).

Using LXDE (Lightweight X11 Desktop Environment)

The Lightweight X11 Desktop Environment is the standard windowing system for Raspbian. Its basic setup is the same as most versions of Windows pre-Windows 8. There's a button in the bottom-left side of the screen that opens an applications menu, and currently running applications are displayed in the bar along the bottom (see Figure 1-1).

Figure 1-1: The LXDE desktop with the menu open.

If you get a black screen with white text asking you to log in when you boot up your Pi, it means that you haven't set it up to start LXDE automatically. Don't worry; just log in with the username pi and the password raspberry, and then type the following:

startx
You can set it up to boot into LXDE automatically using raspi-config (see the next section).

Using the Terminal

LXDE is great for many tasks, but sometimes you'll need to use the command line. This is an incredibly powerful interface that's accessed through the terminal. In LXDE, that means opening the LXTerminal application.

When you open LXTerminal, you should see the following line:
pi@raspberrypi~$
This signifies that you are using the username pi on a computer called raspberrypi, and you are in a directory called ~.

In Linux, all directories start from / or root. This is the base of the directory tree and every directory is located in some subdirectory of this. You can move between directories using the cd (change directory) command. Start by moving to this root directory with:

cd /
You should now seen that the command prompt has changed to
pi@raspberrypi/$

You can list the contents of this directory with the command ls. One of the subdirectories is called home. This is where every user on the system has his home directory. Move into it and view its contents with:

cd home ls
There should only be one directory called pi. The command prompt should now have changed to show that you're in the directory /home. Move into the only subdirectory with:
cd pi
Now the command prompt will have reverted to:
pi@raspberrypi~$
This is because the character ~ is a shorthand for the current user's home directory. When you type ~ in the terminal, the computer converts it to /home/pi.

There is much more to learn about the command line. So much so that it would take another book this size to cover it with any semblance of completeness. However, you don't need to know everything to start using it, and whenever we

tell you to use LXTerminal, we tell you exactly what to type.

TIP If you are interested in learning more about the Raspberry Pi, or Linux in general, the command line is an excellent place to start, and there's loads of information about it both online and in print. The Linux command-line book, which you can browse for free online, is an excellent place to start. See http://linuxcommand.org/tlcl.php.

We'll leave you with two pieces of advice. Firstly, don't be afraid of the terminal. It can be a bit daunting at first, but the only way to learn how to use it is to use it. Secondly, almost all commands have built-in help that you can access using the flag —help. For example, if you want to learn more about how to use the command ls, you can enter:

ls --help
This will output:

Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default). Sort entries alphabetically if none of -cftuvSUX nor
--sort is specified.

It then goes on to list all the various flags you can use with the command.

Changing Configurations with Raspi-Config

Raspbian comes with a tool to help you set up the hardware on your Raspberry Pi; it's called raspi-config. To start it, open LXTerminal and type:
sudo raspi-config

Here, you'll find options to start LXDE automatically when you boot up, overclock your Pi, and other things. Overclocking your Pi will make a few things in this book run a little better, most notably, installing new software.

Installing Software

You can install new software on your Raspberry Pi using the apt-get command in the terminal. Before installing anything, it's a good idea to update all your software to the latest version. You can do this with:

sudo apt-get update sudo apt-get upgrade
Then you can use apt-get to install whatever you want. For example, if you want to use iceweasel (a re-branded version of Firefox), you need to open LXTerminal

and type:
sudo apt-get install iceweasel
If you prefer to do this using a graphical program, you can get the program synaptic with:
sudo apt-get install synaptic
When you want to install something, you can start it with:
sudo synaptic
From there you'll be able to search for whatever you want.

NOTE Whenever you install software, you need to use the word sudo before the command. It tells the computer that you want to make a system-wide change and gives the program sufficient permissions to do this.

Python 3

In this book, you'll learn how to use the Python 3 programming language. In Raspbian, there are a couple of ways to use this language.

The Python Interpreter

There are two ways of using Python, from the shell and saved programs. The shell executes each instruction as you type it, which means it's a really good way of trying out things and doing experiments. Saved programs are bits of Python code that are saved in a text file and run all at once. It's easy to tell which environment you're in because in the shell, all the lines will start with three chevrons:

```
>>>
```

Most of the time in this book, we'll deal with saved programs, but there are some occasions (particularly early on) when we tell you to use the shell. To make it clear which bits of code are for which, we've started every bit of code for the shell with three chevrons.

Running Python Programs

There are two different ways you can write programs for Python. You can create text files that contain the code, and then run these files with Python, or you can use an Integrated Development Environment (IDE) such as IDLE 3. Either way will result in the code being run in the same way and it's just a matter of personal preference.

If you want to write the programs as text files, you need to use a text editor such as Leafpad. A word processor such as LibreOffice's Writer is unsuitable because the various formatting it uses will confuse Python. As an example, open Leafpad and create a new file that just has the line:

```
print("Hello World!")
```

Once you've created your file, just save it with the extension .py; for example testfile.py. You can then run it by opening LXTerminal and navigating to where the file is saved. Then you run python <filename>. You can use the cd command to move to different directories. For example, if you save the file in a folder called programming in your home directory, you could run it by typing the following into LXTerminal:

```
cd programming
python3 testfile.py
```

If everything has worked correctly, you should see the following line appear on the screen:
Hello World!

The second way is a little simpler. Using an IDE, the text editor and Python interpreter are in the same program. For example, open IDLE 3 (make sure to use the one with the 3), and go to File➪New Window. In the new window, enter this code:

```
print("Hello IDLE")
```

Then go to Run➪Run Module. It will prompt you to save the module, so select a filename. Once you've done this, it will switch back to the Python interpreter and display the following:
Hello IDLE It doesn't really matter which one you use, so just go with the way you feel most comfortable with.

Summary

After reading this chapter, you should understand the following a bit better:

■ You'll need a few extra bits of hardware to get the most out of your Raspberry Pi. ■ Insufficient power is the most common cause of problems.

- If you're having problems, dmesg and syslog are the best places to find out what's going on.
- Raspbian uses the LXDE desktop environment.
- The terminal provides the most powerful way of interacting with the underlying operating system.

- Theraspi-config tool lets you configure your Raspberry Pi.
- Use apt-get to install new software.
- You can run Python either through the interpreter or by running saved programs.

Chapter 2
A Really Quick Introduction to Python

IN THIS CHAPTER, you'll dive right into some code examples. Don't expect to grasp all the details yet. This chapter is meant to give you a taste of programming. You'll learn how to draw on the screen, and even how to make a simple game. Along the way you'll pick up some basic programming concepts, but don't worry if you don't understand every line of every program you create in this chapter. You'll learn more about the details in later chapters.

Drawing Picture with Turtles

It's time to get programming! We strongly recommend that you enter the code into IDLE 3 as you read along, as it will help you understand what's happening. So, without further ado, open IDLE 3, go to File➪New Window, and enter the following:

```
import turtle
window = turtle.Screen() babbage = turtle.Turtle() babbage.left(90)
babbage.forward(100) babbage.right(90)
babbage.circle(10)
window.exitonclick()
```

Then go to Run ➪Run Module or press F5 to execute the program. A dialog will open and ask you to provide a filename. This name can be whatever you want, although it helps if it's descriptive so you'll remember it in the future (we used chapter2-example1.py). Each of these lines is an instruction to Python. Python

goes through them one-by-one and executes them in the order it finds them. The result of the computer following all these steps is to draw a line with a circle on top, as shown in Figure 2-1. You might think the drawing looks like a lollipop, but actually, it's the first part of a flower. If you don't get this result, go back and check that everything is typed correctly and try again.

Figure 2-1: Your first turtle drawing with Python.

Let's take a closer look at what Python's doing as it goes through the code.

```
import turtle
```

You’ll often see several import lines at the start of Python programs. They bring extra features into programs a bit like add-ons or plug-ins in other software. These features are grouped into modules. You’ll learn more about how to use the import command in the following chapter. This time we’re importing the turtle module, which lets us draw graphics. The next portion of the code is

window = turtle.Screen() ...
window.exitonclick()

This creates a new window that we can draw onto, and set it to close when it’s clicked.
The next line uses the turtle module that you imported in the first line to create a new turtle, named babbage (after Charles Babbage, who invented the concept of the computer):
babbage = turtle.Turtle()
Babbage has a number of methods, or things you can tell it to do. For example, in the line:
babbage.left(90)

You’re using the method left() which turns babbage left a certain number of degrees. Parameters are added in the brackets after the method and you can use them to send certain bits of information to control how the method runs. In this case, you passed the parameter 90, so babbage turns left 90 degrees. The following lines use the methods forward(), right(), and circle().

babbage.forward(100) babbage.right(90) babbage.circle(10)

The first method moves the turtle forwards 100 pixels, the second turns it right 90 degrees, and the final one draws a circle with a radius of 10 pixels.
Now it’s time to add a petal. Edit the code so that it reads as follows (changes are in bold):

import turtle
#create window and turtle window = turtle.Screen() babbage = turtle.Turtle()
#draw stem and centre babbage.left(90)
babbage.forward(100) babbage.right(90)
babbage.circle(10)
#draw first petal
babbage.left(15)

babbage.forward(50) babbage.left(157) babbage.forward(50) #tidy up window
window.exitonclick()

Run this now. You should see that the flower now has a solitary petal. You'll notice that we've added some lines that begin with a # symbol. The computer ignores any line that starts like this, so you use them to leave little comments to yourself (or anyone else who looks at the code). These make the code more readable, and mean that if you come back to the code in a few days, weeks, or even years, you can easily see what it's doing.

Using Loops

You should find the section that draws the petal quite easy to understand (we calculated the angles of the two left turns using trigonometry, but don't worry, we won't be going into the maths here).

You could now add a block of code to draw a second petal (there'll be 24 in total). It will be exactly the same as the first petal, and directly below it, so with a bit of copy and paste, you'll get the following:

#draw second petal babbage.left(15) babbage.forward(50) babbage.left(157) babbage.forward(50)

And then do the same for the next 22 petals … Okay, hang on here. As a general rule of thumb, when programming, you should never repeat identical code. Suppose you decided to change the size of the petal; you'd have to change it 48 times (twice for each petal), and if you forgot any one, you'd get a wonky picture. Instead, you can use a loop, which is a piece of code that tells the computer to repeat a certain section of code over and over again.

Instead you can replace all the code from #draw first petal downwards with the following code:

#draw all petals
for i in range(1,24): babbage.left(15)
babbage.forward(50)
babbage.left(157)
babbage.forward(50) window.exitonclick()

We'll deal with exactly what the first line (after the comment) means in the next chapter, but for now, it's enough to say that it repeats the block 24 times, then the computer moves on to the instructions after this block.

Code blocks like loops (and others that you'll explore later) always follow the same layout in Python. The first line ends with a colon, and every line after it is indented. Once the tabbing/ indention stops, Python considers this code block over. If you've programmed in other languages before, you'll probably have noticed that they do things a little differently.

Try running the code. You should find that babbage runs round drawing all the petals, and we finish with a complete flower, as shown in Figure 2-2.

Figure 2-2: Loops make drawing flowers a breeze.

Not bad for just 13 lines of code! Of course, not many flowers are all black, so it would be better if you could add a little colour to the picture. The Python turtle module does include some methods that allow you to change the colour of your drawing. Amend the first half of your program so it reads as follows (changes are shown in bold):

import turtle

```
#create window and turtle
window = turtle.Screen()
babbage = turtle.Turtle()
#draw stem

babbage.color("green", "black") babbage.left(90)
babbage.forward(100)
babbage.right(90)
#draw centre
babbage.color("black", "black") babbage.begin_fill()
babbage.circle(10)
babbage.end_fill()
```

As you can see, we're using the color(colour1, colour2) method (Brits should notice the American spelling of the method), where colour1 is the pen colour and colour2 is the fill colour. When you start the centre circle of the flower, you tell the computer to fill in the circle with the begin_fill() method. Afterwards, we used end_fill() so it doesn't keep filling in all the petals.

Conditionals: if, elif, and else

Now type the second half of the flower-drawing program into IDLE 3:

```
#draw all petals
for i in range(1,24):
if babbage.color() == ("red", "black"):

babbage.color("orange", "black")
elif babbage.color() == ("orange", "black"):
babbage.color("yellow", "black")
else:
babbage.color("red","black"))
babbage.left(15)
babbage.forward(50)
babbage.left(157)
babbage.forward(50)
#hide the turtle
babbage.hideturtle()
#tidy up window
window.exitonclick()
```

We’ve used a little artistic licence and decided that the flower should have petals with three different colours: red, orange, and yellow. As this book is in black and white, you’ll have to run the program on your Raspberry Pi, or you can take a look at flower.png on the companion website, to see the result in living color. To alternate our petal colours, we’ve used an if .. elif .. else block. This is a way of telling Python how to make decisions about what to do based on certain data. The basic structure is as follows:

if <condition> : code
where <condition> is a statement that can be true or false. In this case, we’re using the following condition:
babbage.color() == ("red", "black")

babbage.color() (note that now this method doesn’t have any parameters) tells the program what colour our turtle currently has. This is a little different to the methods you’ve seen so far because it sends information back that you can use. This return value is a pair of colours—the first is the drawing colour, and the second is the fill colour (which hasn’t changed since you set it to draw the centre of the flower, so it will stay the same for the whole of the program). The double equals sign (==) means ‘is equal to’. You use a double equals here because a single equals is used differently, like when you created the window and the turtle.

If the condition is true (in this case, if the turtle’s colour is (“red”, ”black”)), then Python executes the code. However, if the condition is false, Python moves on to the elif (elif is short for else if). This has the same structure as the original if condition.

If the condition in the elif is false, then Python moves on to the else. If it gets this far (that is, if the conditions for the if and elif are both false), Python will execute the code. else doesn’t have a condition. Figure 2-3 shows the flow of this logic.

Start
Is the colour red? Yes Set the colour orange
No
Is the colour orange? Yes Set the colour yellow
No
Set the colour red

Figure 2-3: The flow of conditional logic for determining the colour of each flower petal.

This if clause then, will alternate the colour of the pen after each petal is drawn. The final alteration is to add the following line to the end of the program:

babbage.hideturtle()

This simply makes the turtle (cursor) invisible so it doesn't obscure our picture. There you have it; your very first Python program finished completely!

Using Functions and Methods to Structure Code

Before we dive in and start our second Python program, we're going to pause for a second to talk a bit more about methods. As you've seen, methods are really useful for controlling parts of our program. In the previous example, you used them to move turtles, change colour, and create windows. Each time, you called them on something. For example, you called forward(50) on babbage with babbage.forward(50) and exitonclick() on window withwindow.exitonclick(). Each time, the methods run bits of code that are stored in the Python modules. Python has another similar feature called functions. These work in a fairly similar way, but they're not called on anything. For example, in your Python interpreter, type:

```
>>> print("Hello World")
```

This runs the print() function that simply outputs its parameter to the screen. Remember when we said that you shouldn't repeat any code in your programs? We explained that loops are one way of reducing repetition, and functions are another. As an example, think of a program that deals with circles and often needs to calculate the area for a given radius. If you listened in maths classes, you should know that the area of a circle is 2 × pi × the radius (if you didn't listen in maths class, then you'll just have to take our word for it). Rather than repeat this every time you need to calculate the area of a circle, which could lead to problems (if you mistype the value of pi somewhere, it could cause all sorts of problems and be hard to find), you can create a function to do it. In Python this would be:

```
def circlearea(radius): return radius * 3.14 * 2
print(circlearea(1))
```

Here we've used two functions nested together. circlearea(1) calculates the area of a circle with a radius of one, and print() sends this number to the screen.

As you can see, you define your own functions using the word def, followed by the name of the function, followed by the parameters enclosed in brackets. You can then use the name of the parameter inside the function where it will act with the value that's passed to it. The wordreturn tells Python what value you want to send back. So, in the previous example, when Python gets to the phrase circlearea(1), it runs the code under def circlearea(radius), but instead of radius it substitutes the number you passed across (1). Then it returns the value of that calculation (6.28) to the print function. You'll see later that you can nest methods in the same way so that one method sends information straight to another one. This can be a really useful way of getting data to flow in the right way between different sections of your program.

A Python Game of Cat and Mouse

Now, let's move on to our second Python program. This time you're going to make a game of cat and mouse. The player will control the mouse using the arrow keys, and she has to stay ahead of the cat (controlled by the computer). The longer she stays ahead, the higher score she gets.

Most of the longer code examples in the book are available for download from the book's **NOTE**companion website at www.wiley.com/go/pythonraspi. To avoid potential typos, you can
download and copy and paste the text into your IDE or code editor. The code for the following example is Chapter2-catandmouse.py.

Open a new window in IDLE 3 and type the following code:

```
import turtle import time

boxsize = 200 caught = False score = 0

#functions that are called on keypresses def up():
mouse.forward(10)
checkbound()

def left():
mouse.left(45) def right():
mouse.right(45)

def back():
mouse.backward(10) checkbound()
```

```
def quitTurtles(): window.bye()

#stop the mouse from leaving the square set by box size def checkbound():
global boxsize
if mouse.xcor() > boxsize:
mouse.goto(boxsize, mouse.ycor())
if mouse.xcor() < -boxsize:
mouse.goto(-boxsize, mouse.ycor())
if mouse.ycor() > boxsize:
mouse.goto(mouse.xcor(), boxsize)
if mouse.ycor() < -boxsize:
mouse.goto(mouse.xcor(), -boxsize)

#set up screen
window = turtle.Screen() mouse = turtle.Turtle() cat = turtle.Turtle()
mouse.penup()
mouse.penup()
mouse.goto(100,100)

#add key listeners
window.onkeypress(up, "Up")
window.onkeypress(left, "Left")
window.onkeypress(right, "Right") window.onkeypress(back, "Down")
window.onkeypress(quitTurtles, "Escape")

difficulty = window.numinput("Difficulty",
"Enter a difficulty from easy (1), for hard (5) ", minval=1, maxval=5)

window.listen() #main loop
#note how it changes with difficulty
while not caught:

cat.setheading(cat.towards(mouse))
cat.forward(8+difficulty)
score = score + 1
if cat.distance(mouse) < 5:

caught = True
time.sleep(0.2-(0.01*difficulty))
window.textinput("GAME OVER", "Well done. You scored: "+
```

```
str(score*difficulty))
window.bye()
```

That’s quite a lot of code, but before you look at it too deeply, play the game a few times to get a feel for it. This will also help you make sure that you’ve entered it correctly. If you get any errors, check your typing, and then try again. Take a look at Figure 2-4 to see it in action.

Figure 2-4: A simple game of cat and mouse.

Understanding Variables

The first two lines just bring in modules for using turtles and time, and then the following three lines are

```
boxsize = 200
caught = False
score = 0
```

These lines use something that we’ve touched on, but haven’t really talked about: variables. Variables are places where you can store values you want to use

later. For example, in the first line, you store the value 200 in the variable boxsize. After you've set them up like this, you can simply put in boxsize and Python will substitute the correct value. These constructs are called variables because they can change. In this particular program, boxsize will stay the same, but both caught and score will vary throughout it. Each time you want a new value, you simply use the single equals sign. This is the same thing you did in the first example withwindow and babbage; there the variables held the screen and the turtle. We'll cover variables, and what exactly you can store in them in the next chapter.

Defining Functions

The next part of the code defines some functions that you'll use in the program. In the functioncheckbounds(), you'll notice that there's the following line:

```
global boxsize
```

This line is needed because functions don't automatically get access to variables defined outside of them. This line tells Python that we want to use the boxsize variable in this function, and it's declared outside of the function itself.

Perhaps the most confusing section is

```
#add key listeners
window.onkeypress(up, "Up")
window.onkeypress(left, "Left")
window.onkeypress(right, "Right") window.onkeypress(back, "Down")
window.onkeypress(quitTurtles, "Escape")
```

This code tells the window what to do when various keys are pressed. For example, the first line says to run the function up (which we've already defined) whenever the key "Up" (which corresponds to the up arrow on the keyboard) is pressed.

Looping Through the Game

Next you get to the main loop that runs the game:

```
while not caught:
cat.setheading(cat.towards(mouse)) cat.forward(8+difficulty)
score = score + 1
if cat.distance(mouse) < 5:
```

```
caught = True
time.sleep(0.2-(0.01*difficulty))
```

This code uses a different type of loop. The while loop take this form:
while condition: loop code
They keep on looping the code as long as the condition is True. In the initial list of variables at the beginning of the code, you set the variable caught toFalse:
caught = False

Thus in this case, not caught is the condition (and it's true at the start since not False is True), so the program keeps running until you change it to true because not True is False. It all sounds a bit complex when phrased like this, but an easy way to think of it is that the word not just swaps True and False around.

time.sleep() tells Python to stop for a certain number of seconds. In this case you reduce the amount of time it sleeps as the difficulty level (which is a variable set to a number that the user enters) increases. You should also be able to see that the distance the cat moves increases with difficulty.

TIP

At this point you may be wondering how on earth you're supposed to remember all the methods that are associated with the various modules. For example, how did you know to use forward(10) instead of forwards(10) or move_forwards(10), or for that matter, how did you know there was a method to move forwards at all? Well, you don't have to have superhuman memory to use Python; you just need to know where to look. When you're in IDLE 3, press F1 to open your web browser and display the Python documentation. There's some really useful information here and it's well worth exploring. For information on the turtle module, just enter turtle into quick search, and then select the top response. You'll see that we've only touched on its methods here.

Summary

This brings us to the end of our really quick tour of Python. Hopefully the programs made some sense to you. Don't worry if you didn't understand a hundred percent of everything, because we're going to look at the different parts of Python in a bit more detail in the next chapter. However, hopefully you now understand the following:

■ Python programs consist of a series of instructions and they run from top to bottom. ■ You can control the way Python moves through your program using loops and if statements.
■ You don't have to do everything yourself. You can import modules and use methods to take care of much of the work.
■ Functions allow you to reuse code, which makes your programs easier to understand and maintain.
■ Variables allow you to store information so you can use it later.
■ It's really easy to draw flowers and make games in Python.

Remember, when programming there's often more than one way to do something, and if you can pick the right way (or at least, not the wrong way) you'll make your life easier. However, it's not always easy to know if a way is right or wrong, so we'll finish this chapter with Python's own advice on the matter. In your Python interpreter, type:

```
>>> import this
```
this is a special module that outputs some useful Python advice when it's imported. Now that you have a feel for Python, let's move on and dig into the code a bit deeper.

Chapter 3
Python Basics

IN THE PREVIOUS chapter, we got straight into programming in Python. Hopefully that gave you a good idea of what Python is, but you probably have quite a few questions about how it all worked. In this chapter, we'll aim to answer those questions, and go into detail about exactly how to create your own programs in Python. Then, in later chapters, we'll look at the specific features of Python that help you write particular types of programs for the Raspberry Pi.

Variables, Values, and Types

In the last chapter, we saw that variables can store data we want to use elsewhere. They're one of the most powerful tools programmers can use, and now we'll look at what they actually are. If you've programmed before in a different language, you may notice that Python does things a little differently here.

In the Python interpreter, enter the statement:
>>> score = 0

All this does is tell Python that you want to use score as a new name for the value 0. After this point, whenever Python sees score, it will insert the value 0. To demonstrate this, try entering the following:

>>> print(score)
Remember that Python executes our instructions in order, and you must give score a value before you use it. Otherwise, Python will produce an error.
If you want score to point to some other value, you just assign it a new one, such as:
>>> score = 1
Now, Python will substitute 1 every time it sees the word score (you can verify this by running print(score) again). You can update a variable using its own value:
>> score = score + 1

Variables can have almost any name you want, but they must start with a letter or an underscore, and it can't be the same as a word that's used elsewhere in Python (such as if, for, and so on). The convention in Python is to name variables in lowercase and to separate individual words with an underscore, like so:

high_score = 1000
In the previous examples, the values have all been numbers, but this doesn't have to be the case. They can also be text, such as:
player_name = "Ben"
We can even switch a name between numbers and text, such as:

>>> our_variable = 1000
>>> print(our_variable)
>>> our_variable = "Some Text" >>> print(our_variable)

However, the value that a variable currently points to has a particular type.
Values Have Types

When you see a number 3, you probably just see that, a 3. It doesn't matter if it's in a sentence or a sum, 3 always means 3. Python, however, sees it differently. Every piece of data has to have a type associated with it so it knows what it's dealing with. You can find out what type Python has associated with a bit of data

by using the type() function. In a Python interpreter, enter:

```
>>> type(3) <class 'int'>
>>> type("3") <class 'str'>
```

The first one, Python will inform you, is an int (short for integer—basically a whole number), whereas the second is a str (short for string—a piece of text). That is because Python sees the number three differently from the character 3. To see the difference, enter these two lines:

```
>>> 3+3 6
>>> "3" + "3" 33
```

The first will add the two numbers together whilst the second will join the two pieces of text together. As you can see, it’s vitally important that you don’t mix up your types or you could get some pretty interesting results. To discover some more types, enter:

```
>>>type(3.0) <class 'float'>
>>>type(3>2) <class 'bool'>
```

The first will give you float (a floating-point number, which is any number with a decimal point). The second gives you bool (a Boolean, which can take just one of two values: True and False).

Storing Numbers

The type of a particular piece of data affects what Python can do with it. We’ll start with numbers (both int and float types, but not strings even if they contain numbers). There are two types of operations we can do with numbers: comparisons and numerical operations. Comparisons, as listed in Table 3-1, take two numbers and return a bool.

Table 3-1 Comparison Operators for Numerical Types

Operator Meaning Example < Less than 3<2 → False > Greater than 3>2 → True == Equal 3=3 → True <= Less than or equal to 3<=3 → True >= Greater than or equal to 3>=4 → False != Not equal to 3!=4 → True

Meanwhile, numerical operations return a numerical data type and are shown in Table 3-2.

Table 3-2 Numerical Operators
Operator Meaning Example + Addition 2 + 2 → 4
- Subtraction 3 − 2 → 1 * Multiplication 2 * 3 → 6 / Division 10/2 → 5 %
Divide and take the remainder 5%2 → 1 ** To the power 4 ** 2 → 16 int()
Convert to integer int(3.2) → 3 float() Convert to float float(2) → 2.0

You can test any of these operators by typing them into the Python interpreter. For example:
>>> 3!=3 False
If you're using numerical operators in your program, you'll usually want to capture the value they return in a variable. For example:
>>> number_1 = 10
>>> number_2 = number_1** 2

Keeping Text in Strings

The string type allows you to store any chunks of text that you need. To create a string, simply enclose the data in either single or double quote marks. As far as Python is concerned, either type of quotation is fine. We prefer to use double quotes because this doesn't cause problems when creating strings that have apostrophes in them, but this is by no means universal and some programmers prefer single quotes because they're easier to type.

This data type is a little different from the others because, in many ways, a string isn't a single piece of data but a collection of letters. The name comes from the fact that they are a string of characters.

As with numbers, Python gives us operations we can perform on them. Table 3-3 shows a few of the most useful.

Table 3-3 String Operations Operator
string[x]
string[x:y] string[:y]
string[x:]
len(string) string+ string

Meaning Example
Get the xth character (starts from 0th) "abcde"[1]→ "b" Get all the characters from the xth to the yth "abcde"[1:3]→ "bc" Get every character up until the yth "abcde"[:3]→ "abc" Get every character from the xth to the end "abcde"[3:]→

"de" Return the length of the string len("abcde")→ 5 Join two strings together "abc"+"def"→"abcdef"

Boolean: True or False

The last data type that we'll look at here is bool. This is the simplest because there are just two values it can take: True and False. Note that in Python, these values must always have a capital first letter and aren't surrounded by any quotation marks. Although this data type isn't often stored in variables (although it can be as we saw last chapter), it is often used in conditions forif statements and loops, which we'll cover in more detail a bit later in this chapter. The main operators on this are and, or, and not.

not simply switches the value, so:
>>> not True False
>>> not False True
and takes two bools and returns True if they are both True, and False otherwise, so:
>>> True and False False
>>> False and False False
>>> True and True True
or also takes two bools, but it returns True if either one of them is True, so:
>>> True or False True
>>> True or True True
>>> False or False False

Converting Between Data Types

It is possible to convert between data types using the functions int(), float(), and str(). They convert other data types to integers, floating-points, and strings, respectively. However, these can't just change anything to anything. If you change a float to an int, Python will round down any fractions. Strings can be converted to numbers only if they contain only digits. On the other hand, more or less anything can be converted into a string. Take a look at the following examples:

>>> print(int(3.9)) 3
>>> print(str(True)) True
>>> print(float("Three point two"))
Traceback (most recent call last):

File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'Three point two'

Test Your Knowledge

Following are a series of Python statements. See if you can work out what each of them means to Python. When you think you know, type them into a Python interpreter to check. Hint: some will produce errors.

■ int("two") ■ print(str(3+3) + "3") ■ type(3=3) ■ "Python"[4] ■ (3 > 2) or (2 > 3) ■ not ”True” ■ 2345[2] ■ str((not True) and (not False)) ■ 10 % 3

■ "4" == 4

Exercise 1
What are the variables, what values do they hold, and what are the types of these values in the following program? The answers are at the end of the chapter. If you’re unsure, try typing them into Python and inserting print statements in various parts of the code to see what’s going on. This is an excellent way of finding out what a particular piece of code is doing when it’s not running as you expect.

```
prompt_text = "Enter a number: " user_in = input(prompt_text) user_num =
int(user_in)

for i in range(1,10):
print(i, " times ", user_num, " is ", i*user_num)
even = (user_num % 2) == 0

if even:
print(user_num, " is even")
else:
print(user_num, " is odd")
```

Storing Values in Structures

As well as the simple data types, Python allows us to bring data together in various ways to create structures. The simplest structures are called sequences. These store information one piece after the next. There are two sorts: lists and tuples. In many ways, they’re very similar. Take a look at the following example:

```
>>> list_1 = [1,2,3,4] >>> tuple_1 = (1,2,3,4) >>> list_1[1]
2
>>> tuple_1[1]
2
```

You create a list by enclosing it in square brackets, whilst tuples are surrounded by round ones. So far, they seem to be working the same. You can retrieve an individual item from either by putting square brackets after it with an index. Note that indexes start at 0, so list_1[0] and tuple_1[0] will retrieve the first item in either sequence.

The difference between lists and tuples comes when you try to update them:

```
>>> list_1[1]=99
>>> list_1
[1, 99, 3, 4]
>>> tuple_1[1]=99
Traceback (most recent call last):
File "<pyshell#35>", line 1, in <module>
t1[1]=99
TypeError: 'tuple' object does not support item assignment
```

As you can see, you can update an item in a list, but not in a tuple. However, whilst you can’t update a tuple, you can simply overwrite the lot. In this case, you’re simply telling Python to point the variabletuple_1 to a new tuple and discard the old one.

```
>>> tuple_1 (1, 2, 3, 4)
>>> tuple_1=(1,99,2,4)
>>> tuple_1 (1, 99, 2, 4)
```

In the previous section we covered strings. These are just sequences of characters, and the operations you can perform on them you can also perform on lists and tuples. For example, continuing to use list_1 and tuple_1:

```
>>> len(list_1) 4
>>> tuple_1[:3] [1, 99, 2]
```

You can flick back to Table 3-3 for a recap on the string operators we can use here. The individual items in lists and tuples can be any data type, including lists and tuples themselves. If you so wish, you can have a list of lists of lists of lists;

although if you do, the chances are that your code is going to be pretty hard to understand. Having a list of lists, however, can often be very useful. You can think of this a bit like a table, so:

```
>>> list_2 =
[["a","b","c"],["d","e","f"],["g","h","i"],["j","k","l"]]
```

You can get items out of it by indexing both the main list and the sublist:

```
>>> list_2[2][0] 'g'
>>> list_2[0] ['a', 'b', 'c']
>>> list_2[0][1:] ['b', 'c']
```

Take a look at Table 3-4 to see how this corresponds to a table.

Table 3-4 **An Illustration of a Two-Dimensional List**

list_2[0][0] list_2[1][0] list_2[2][0] list_2[3][0] = "a" = "d" = "g" = "j" list_2[0][1] list_2[1][1] list_2[2][1] list_2[3][1] = "b" = "e" = "h" = "k" list_2[0][2] list_2[1][2] list_2[2][2] list_2[3][2] = "c" = "f" = "i" = "l"

There are a few operations that we can carry out to manipulate lists. Some of the most useful are listed in Table 3-5.

Table 3-5 **Operations on Lists (in All the Examples, Use**

Operator	Meaning	Example
list.append(item)	Add item to the end of the list	list_3.append(0) → [3,2,1,0]
list.extend(list_2)	Join list_2 to the end of list	list_3.extend([0,-1]) → [3,2,1,0,-1]

list_3 = [3,2,1])

continued

Table 3-5 **continued** Operator

list.pop(x)

list.insert
(x, item)
list.sort()

list.index(item)
list.count(item)
list.
remove(item)

Meaning
Return and remove the xth item Insert item at the xth position

Sort the list

Return the position of the first occurrence of item in list
Count how many times item appears in list

Remove the first occurrence of item in list

Example
See below

list_3.insert(99,1) → [3, 99, 2, 1]
list_3.sort() → [1,2,3]

List_3.index(2) → 1
list_3.count(2) → 1
list_3.remove(2) → [3,1]

Most of these examples are a little different to the ones we've given before, because each of them (exceptindex() and sort()) changes the value of list_3 rather than returning anything. So, for example, to run the first one in the Python interpreter, you'll also need a line to show the final value of list_3.

>>> list_3 = [3,2,1] >>> list_3.append(0) >>> list_3
[3, 2, 1, 0]

index()

and count(), however, return a value, so:
>>> list_3.index(2) 1

pop(x) is a little unusual as it does two things. Firstly, it returns the value at the xth position, but it also removes it from the list. Try out the following example to get a feel for what it does:

>>> list_3 = [1,2,3] >>> out = list_3.pop(1) >>> out
2
>>> list_3
[1, 3]

As mentioned, tuples are a lot like lists except that they can't be changed. Any of the list operators that don't change the value can also be used on tuples, so:

>>> tuple_2 = (1,2,3)
>>> tuple_2.index(2)
1
>>> tuple_2.sort()
Traceback (most recent call last):

File "<pyshell#91>", line 1, in <module>
tuple_2.sort()
AttributeError: 'tuple' object has no attribute 'sort'

Non-Sequential Values in Dictionaries and Sets

You can think of lists and tuples as groups of items that each have a number associated with them as an index. For example, in the list ["a", "b", "c", "d"], a has the index0, b has the index1, and so on. However, what if you want to use an index that isn't a number? For example, perhaps you wanted to create a data structure that linked your friends' nicknames to their real names. It could work like this:

>>> real_name["Benny"] 'Benjamin Everard'
In Python, you can do this using dictionaries, and you create them using curly braces. You can create the real_name dictionary using:
>>> real_name = {"Benny" : "Benjamin Everard", "Alex" : "Alexander Bradbury"}

Items in a dictionary are known as key/value pairs, where the first part (in this case, the nickname) is the key and the second part (the full name) is the value. You can add a new item to a dictionary by simply specifying a new key and giving it a value:

```
>>> real_name["Liz"] = "Elizabeth Upton"
>> real_name
{'Alex': 'Alexander Bradbury', 'Benny': 'Benjamin Everard',

'Liz': 'Elizabeth Upton'}
```

You may be wondering why you need indexes or keys at all. In fact, you don't. Python also allows you to just lump a load of data together with no indexing or ordering using the set structure. For example:

```
>>> herbs = {'thyme', 'dill', 'corriander'} >>> spices = {'cumin', 'chilli',
'corriander'} >>> "thyme" in herbs
True
```

As you can see, Python has the in operator to test if a particular value is in a set. There are also a few operators that are specific to sets. Take a look at Table 3-6.

Table 3-6 Operations on Sets (Examples Use the Sets Previously Defined)

Operator	Meaning
set_1 & set_2	Return the items that are in both sets
set_1 | set_2	Combine the items in both sets
set_1 - set_2	The items in set_1 that aren't in set_2
set_1 ^ set_2	The items that are in set_1 orset_2, but not both

Example

herbs & spices → 'corriander'

herbs | spices → {'dill', 'thyme', 'chilli', 'corriander', 'cumin'}
herbs – spices → {'dill', 'thyme'}

herbs ^ spices → {'dill', 'cumin', 'thyme', 'chilli'}

Test Your Knowledge

What do you think the following will do? Try to work it out, then enter them into the Python interpreter to find out. Remember, some of them will produce errors.

■ ["a", "b", "c"].index("c") ■ (3,2,1).pop(2) ■ {1,3,5} & {2,4,6} ■ {1,2,3} & {1} ■ 3 in {1,2,3} ^ {3,4,5} ■ "abcde".remove("c") ■ 3 not in (1,2,3)

Controlling the Way the Program Flows

while loops are the simplest kind of loop. They have a condition that can be anything with a bool data type, and they will continue looping until that condition is False. If that condition never becomesFalse, then they will simply keep looping forever. For example:

```
>>> while True:
print("Ben is awesome")
```

Hopefully, you'll remember this code block from the previous chapter. There's a colon after the condition, and then the next line is indented. Anything that is indented is considered part of the code block. To run this in the Python interpreter, you'll need to press Return after the print line, then backspace to remove the automatic tab, then Return once more. This lets Python know that the code block has finished and you want to execute it.

If you find yourself stuck in an infinite loop like this, you can stop it by pressing Ctrl+C.

The condition can be as complicated as you like, as long as it comes back to either True or False. However, in order for it to eventually terminate, it will have to include one or more variables that can change inside the loop. For example, take a look at the main loop in the following number-guessing game.

```
import random
secret = int(random.uniform(0,10))
print("I'm thinking of a number between zero and ten."

, "Can you guess what it is?")
guess = 11
while guess != secret:
guess = int(input("Take a guess: "))
```

```
print("Well done!")
```

Note that this program will throw an error if you enter text that isn’t a number. We’ll look at how to fix that shortly, but first we’ll take a look at a different type of loop.

Moving Through Data with for Loops

for loops are used to move through pieces of data, and perform the instructions in the loop on each piece of data in turn. These are commonly used with range(x,y), which moves through every number between x and y. For example, you can calculate the 12 times table with:

```
>>> for i in range(1,13):
print(i, " times twelve is ", i*12)
```

range() can take another parameter that sets the gap between two numbers. For example, if you change range(1,13) torange(2,14,2) it will go through all the even multiples of 12. We can also use for loops to move through any of our sequence data types (including strings) and sets, so the following are also all valid:

```
>>> for i in [1,2,3,4,5,6,7,8,9,10,11,12]: print(i, " times 12 is ", i * 12)
>>> for i in (1,2,3,4,5,6,7,8,9,10,11,12):
print(i, " times 12 is ", i*12) >>> for i in {1,2,3,4}:
print(i, " times 12 is ", i*12) >>> for i in "123456789":
print(i, " times 12 is ", int(i)*12)
```

Although the last one is a pretty good example of using the wrong data type for a task. The set is a little different from the others because you can’t specify the order—it’s just a bunch of items. The following will produce the same result.

```
>>> for i in {1,2,3,4}:
print(i, " times 12 is ", i*12)
>>> for i in {4,3,2,1}:
print(i, " times 12 is ", i*12)
```

Dictionaries are the odd data type out because they don’t just hold items, but key/value pairs. You can still use for loops to move through them, but you have to do it in a slightly different way:

```
>>> real_name = {"Benny": "Benjamin Everard", "Alex": "Alexander Bradbury"}
>>> for key,value in real_name.items():
print("The real name of " + key + " is " + value)
```

As with sets, you can’t specify the order that the loop goes through the data in. In fact, caring about the order the loop goes through the data is a pretty good indication that you should be using a sequence data structure such as a list or tuple rather than an unordered data structure like a set or dictionary.

In the previous examples, i, key, and value are just variable names and can be changed to whatever you want.

Going Deeper with Nested Loops

When programming, you often have to go through more than one thing at a time, such as in the following program, which calculates all the prime numbers between 1 and 30.

```
for i in range(1,30):
is_prime = True
for k in range(2,i):

if (i%k) == 0:
print(i, " is divisible by ", k) is_prime = False

if is_prime:
print(i, " is prime ")
```

Note how the indent level is increased, so inside the first loop there is one indent, then inside the second loop there are two. This is important so Python knows which bits of code belong to which code blocks and where each block ends.

When nesting loops like this, you do need to be a little careful to avoid slowing down your programs too much. The previous example runs quite quickly, but if you try to calculate all the prime numbers under 3,000 (simply add an extra two zeros to the 30 in the first line) then it takes much longer. Not only because there are a hundred times as many outer loops, but as you get into the higher numbers, each inner loop takes longer as well. The whole thing quickly grinds to a crawl (if you try this out, remember Ctrl+C will stop the program). Fortunately, there are a couple of things you can do. Try out the following:

```
for i in range(1,3000,2):
is_prime = True
for k in range(2,i):

if (i%k) == 0:
print(i, " is divisible by ", k) is_prime = False
break

if is_prime:
print(i, " is prime ")
```

The first thing this does is skip over all the even numbers using range(1,3000,2). Of course, this will miss the first prime number, but we already know that one. Straight away, this saves half the time. The second thing we did, though, saves far more. You'll notice the line containing just break in the if block. Since we're calculating prime numbers, we don't care about all the factors of a number, so once we've found one we know that the number's not prime. Thebreak statement then ends the current loop, in this case the inner one, and continues execution from the bottom (at the line if is_prime:). These two optimisations allow the program to run much faster.

Branching Execution with if Statements

As well as looping round sections of code, you can also control the flow of a Python program by branching off and executing different pieces of code depending on a condition. This is done using the if statement that you've already seen quite a few times, but we'll briefly recap it. if statements, like while loops, take a condition that has to have the bool data type. They can also have additional elif (else-if) statements, and an else statement. For example:

```
num = int(input("enter a number: "))
if num%2 == 0:
print("Your number is divisible by 2")

elif num%3 == 0:
print("your number is divisible by 3")
elif num%5 == 0:
print("your number is divisible by 5")
else:
print("your number isn't divisible by 2,3 or 5")
```

if statements only execute at most one block. Once Python finds a condition that is True, it'll execute that block and then exit the if statement. So if you entered 10 as your number, it will only say it's divisible by 2, and won't test if it's divisible by 5 (if you wanted to do that, you'd need to use a separate if block, not an elif statement). The else block is executed if none of the conditions areTrue. As you've seen in other examples, if blocks don't have to haveelif orelse statements. If there's no else statement, and none of the conditions are True, then it simply exits the if block without executing anything.

Catching Exceptions

If you happen to be a mischievous sort of person, you may have noticed that the previous example will throw an error if you enter anything other than a digit. This is because Python can't convert arbitrary strings into numbers and so doesn't know what to do. Whenever Python doesn't know what to do, it throws an exception. So far, we've been letting these go, and when they do, they crash the program.

However, if you know that a particular section of code is likely to throw an exception, you can tell Python to look out for it, and let the program know what to do if it encounters such a problem. This is done in the following way:

```
try:
code where there might be a problem
except type_of_error:
code to run if there's an error
```

You can tell what type of error it is, because Python will tell you. For example:

```
>>> num = int(input("Enter a number: ")) Enter a number: dasd
Traceback (most recent call last):

File "<pyshell#176>", line 1, in <module>
num = int(input("Enter a number: "))
ValueError: invalid literal for int() with base 10: 'dasd'
```

Here you can see it's a ValueError, so the previous example becomes:

```
is_number = False num = 0

while not is_number:
```

```
is_number = True
try:

num = int(input("enter a number: ")) except ValueError:
print("I said a number!")
is_number = False

if num%2 == 0:
print("Your number is divisible by 2")
elif num%3 == 0:
print("your number is divisible by 3")
elif num%5 == 0:
print("your number is divisible by 5")
else:
print("your number isn't divisible by 2,3 or 5")
```

Exercise 2
Try to fix the number-guessing game from the start of this section using a try statement to catch exceptions caused by the users entering bad data.

Making Code Reusable with Functions

We've already used quite a few functions. For example, print() and input(). These standard functions are built into Python, but you can also build your own to perform whatever you want to do. You did this in the previous chapter, so we'll just re-cap it here. Take a look at:

```
>>> def square(num):
return num**2
>>> square(4)
16
```

This code defines a function called square and then uses it to calculate the square of 4. When you use it, you need to enter a single number enclosed in brackets after it—this is known as a parameter. When the function is run, the name that you've given the parameter (in this case num) takes the value of the parameter. Somewhere in the function there can also be a return statement that's used to send data back to the main program. If there's more than onereturn statement, Python will finish executing the function after the first return it reaches.

You can also create functions with more than one parameter. For example, the following program has a function that takes two parameters and returns the largest.

```
def biggest(a,b): if a>b:
return a else:
return b
print("The biggest of 2 and 3 is ", biggest(2,3)) print("The biggest of 10 and 5 is ", biggest(10,5))
```

So far, this all works well, but what about a function that changes the data? Take a look at the following program:

```
def add_one(num): num = num + 1 return num

number_1 = 1
number_2 = add_one(number_1)
print("number_1: ", number_1) print("number_2: ", number_2)
```

Before you run it, think about what you expect it to output. It's pretty clear that number_2 will be 2, but what should number_1 be? Should it be 1 because we set it to 1 in our main program, or should it be 2 because we passed it to add_one() and there it changed?

When you run it, you'll find that number_1 stays as 1. That's because in the line:

```
num = num + 1
```

you're effectively telling Python that you don't want num to have its old value (which was the same as number_1), but to give it a new value that's one larger. However, mutable data types such as lists, sets, and objects can change. If we pass one of these to a function and change it, the original will change. Take a look at:

```
def add_item(list_1): list_1.append(1) return list_1

list_2 = [2,3,4]
list_3 = add_item(list_2) print("list_2: ", list_2) print("list_3: ", list_3)
```

If you run this, you'll get:

```
list_2: [2, 3, 4, 1] list_3: [2, 3, 4, 1]
```

Usually, this shouldn't cause a problem, but occasionally you may come across a situation where you don't want this to happen. For these cases, you can use

copy.deepcopy(), although first, you'll have to import copy. So, if you change the previous example to:

```
import copy
def add_item(list_1): list_1.append(1)
return list_1

list_2 = [2,3,4]
list_3 = add_item(copy.deepcopy(list_2)) print("list_2: ", list_2)
print("list_3: ", list_3)
```

You'll get:
list_2: [2, 3, 4] list_3: [2, 3, 4, 1]

Optional Parameters

Sometimes you may want to create a function that has parameters that are sometimes needed, and sometimes not. For example, you may want to create a function increment() that can take two numbers and add them. However, if only one parameter is given, the function just adds one to it. You do this by specifying a default value (in this case 1) for the parameter like so:

```
def increment(num1 = 1, num2): return num2 + num1
```

The only thing you need to know about these is that the parameters that can be omitted must come after the ones that have to be there.

Bringing Everything Together

We've covered a lot of bits so far in this chapter, but it might not be completely clear how to bring them all together to create the programs you want. In this section, you'll see an example that uses most of what we've introduced so far to create a simple database of student results at a college.

It'll load some default data, and allow you to edit it. We'll use appropriate data types and structures to hold the various pieces of information, and functions to perform actions on that data. We'll even include a simple menu to let the user manipulate the data.

NOTE

Most of the longer code examples in the book are available for download from

the book’s companion website at www.wiley.com/go/pythonraspi. To avoid potential typos, you can download and copy and paste the text into your IDE or code editor.

The code is as follows (you can also find it on the book’s website as The code is as follows (you can also find it on the book’s website as student-1.py).

```
students = [["Ben", {"Maths": 67, "English": 78, "Science": 72}], ["Mark",
{"Maths": 56, "Art": 64, "History": 39, "Geography": 55}],
["Paul", {"English": 66, "History": 88}]]
grades = ((0, "FAIL"),(50, "D"),(60,"C"),(70, "B"), (80,"A"), (101, "CHEAT!"))
def print_report_card(report_student = None): for student in students:
if (student[0] == report_student) or

(report_student == None):
print("Report card for student ", student[0]) for subject, mark in
student[1].items():

for grade in grades:
if mark < grade[0]:
print(subject, " : ", prev_grade) break

prev_grade = grade[1]

def add_student(student_name):
global students
for student in students:

if student[0] == student_name:
return "Student already in database" students.append([student_name, {}]) return
"Student added successfully"

def add_mark(student_name, subject, mark):
global students
for student in students:

if student[0] == student_name:
if subject in student[1].keys():
print(student_name, " already has a mark for ",

subject)
```

```
user_input = input("Overwrite Y/N? ") if user_input == "Y" or "y":

student[1][subject] = mark
return "Student's mark updated" else:

return "Student's mark not updated" else:
student[1][subject] = mark
return "Student's mark added"

return "Student not found"

while True:
print("Welcome to the Raspberry Pi student database") print("What can I help
you with?")
print("Enter 1 to view all report cards")
print("Enter 2 to view the report card for a student") print("Enter 3 to add a
student")
print("Enter 4 to add a mark to a student") print("Enter 5 to exit")

try: user_choice = int(input("Choice: "))
except ValueError:
print("That's not a number I recognise") user_choice = 0

if user_choice == 1:
print_report_card()
elif user_choice == 2:
enter_student = input("Which student? ") print_report_card(enter_student)
elif user_choice == 3:
enter_student = input("Student name? ") print(add_student(enter_student))
elif user_choice == 4:
enter_student = input("Student name? ") enter_subject = input("Subject? ")
num_error = True
while num_error:
num_error = False
try:
enter_mark = int(input("Mark? "))
except ValueError:
print("I don't recognise that as a number") num_error = True
print(add_mark(enter_student, enter_subject, enter_mark))
```

```
elif user_choice == 5:
break
else:
print("Unknown choice")
input("Press enter to continue")
print("Goodbye and thank you for using the Raspberry Pi" , "Student database")
```

The students' data structure is currently a list of lists with dictionaries. It could equally be a dictionary of dictionaries, or a list of lists containing lists. Try to alter the program to match each of these two scenarios. Which feels more natural?

Exercise 3
Could the data also be held in tuples? Why or why not? See the end of the chapter for the answer.

Building Objects with Classes

Classes allow you to link together data and functions into a single object. In fact, we used them in the previous chapter. Remember the lines:

```
window = turtle.Screen()
babbage = turtle.Turtle()
```

At the time we skipped over them quite briefly, but now we'll look at what they do. The statementturtle.Turtle() returns a new object made from the Turtle class that's in the turtle module. Likewise, turtle.Screen() returns an object made from the Screen class in the turtle module. Basically, classes are the blueprints from which you can create objects. These objects then hold data and have methods that you can call to manipulate that data. Methods are really just functions that are held inside classes.

You've already seen how objects can be useful. In the examples in the last chapter, you didn't have to worry about keeping track of any of the turtle data because it was all held in an object. You just stored the turtle object in a variable called babbage and whenever you called a method, it knew everything it needed to. This helped you keep the code clear and easy to use. For example, look at the following code:

```
babbage.forward(100)
```

This moved the turtle forwards and drew the result on the screen. It knew what colour to draw, where the turtle was starting from, and a whole myriad of other information that it needed to draw the line on the screen because it was all stored inside the object.

Let’s take a look at what’s in a class with a simple example:

```
class Person():
def _ _init_ _(self, age, name): self.age = age
self.name = name

def birthday(self):
self.age = self.age + 1

ben = Person(31, "Ben") paul = Person(42, "Paul") print(ben.name, ben.age)
print(paul.name, paul.age)
```

There are a few things to notice here. In Python, the normal style is to start all variable, function, and method names with lowercase letters. Classes are the exception, so the Person class starts with a capital P. Python won’t give you any errors if you don’t follow this, but the convention makes it easier to understand other people’s code. You can see that methods are defined much like functions, except that the parameters always start with self. This brings in the local variables. In this example, these local variables are self.age and self.name. These are re-created for each instance of the class. In this example, we create two objects from the class Person (these are known as instances of the class). Each one of these has its own copy of self.age and self.person. We can access these from outside the object (as we have done in the print methods). They’re known as the attributes of the class Person.

There are also two methods. _ _init_ _ is a special method that every class has. It’s called when an instance is created or “initiated”. So, the line ben = Person(31, "Ben") creates a new object from the Person class and calls the _ _init_ _ method with the parameters (31, "Ben"). This sets up the attributes. The second method, birthday(), shows how using classes means we don’t have to keep track of the data outside of these classes. To give a Person object a birthday, just run (for example):

```
ben.birthday()
```

This increases their age by one.

Sometimes, you won’t want to create a class from scratch, but build a new class that’s built upon an existing one. For example, if we wanted to build a class that holds information about parents, they would also have ages, names, and birthdays, so it would be a waste if we had to rewrite this code just for the Parent class. Python allows us to inherit from other classes, such as in the following:

```
class Person():
def _ _init_ _(self, age, name): self.age = age
self.name = name

def birthday(self):
self.age = self.age + 1

class Parent(Person):
def _ _init_ _(self, age, name): Person._ _init_ _(self,age,name) self.children = []

def add_child(self, child): self.children.append(child)
def print_children(self):
print("The children of ", self.name, " are:") for child in self.children:
print(child.name)

john = Parent(60, "John") ben = Person(32, "Ben") print(john.name, john.age)
john.add_child(ben)
john.print_children()
```

Person is the superclass of Parent, and Parent is a subclass of Person. By adding a class name in the brackets after a class definition, it becomes a superclass of the one you’re defining. You can call the _ _init_ _ method of the superclass, and automatically get access to all the attributes and methods of the superclass without having to rewrite the code for the class. This is called inheritance because the subclasses inherit the features of the superclass.

The big advantage of classes is they make it really easy to reuse code. As we saw in the previous chapter, it was simple to manipulate the turtle without worrying too much about how it actually did what it did. Because the turtle class encapsulated everything, you just had to know the methods and you could use it without any problems. Throughout this book, you’ll see how using classes from outside modules makes it really easy to create quite complex programs without worrying about the technicalities of how these things work.

Take a look at the following code, which is a rewrite of the student database using classes to get a feel for how classes work (chapter3-student-2.py on the website).

```
student_data= [["Ben", {"Maths": 67, "English": 78, "Science": 72}],

["Mark", {"Maths": 56, "Art": 64, "History": 39, "Geography": 55}],
["Paul", {"English": 66, "History": 88}]]

grades = ((0, "FAIL"),(50, "D"),(60,"C"),(70, "B"), (80,"A"), (101, "CHEAT!"))

class Student():
def _ _init_ _(self, name, marks): self.name = name
self.marks = marks

def print_report_card(self):
print("Report card for student ", self.name) for subject, mark in
self.marks.items():

for i in grades:
if mark < i[0]:
print(subject, " : ", prev_grade) break

prev_grade = i[1]
def add_mark(self, subject, mark):
if subject in self.marks.keys():
print(student_name, " already has a mark for ",

subject)
user_input = input("Overwrite Y/N? ") if user_input == "Y" or "y":

self.marks[subject] = mark
return "Student's mark updated"
else:

return "Student's mark not updated" else:
self.marks[subject] = mark
return "Student's mark added"

class Students():
def _ _init_ _(self, all_students): self.students = []
```

```
for student, mark in all_students:

self.add_student(student, mark)
def add_student(self,student_name, marks = {}): if self.exists(student_name):
return "Student already in database"

else:
self.students.append(Student(student_name, marks)) return "Student added"

def print_report_cards(self, student_name = None): for student in self.students:
if student_name == None or student.name: student.print_report_card()
def exists(self, student_name):
for student in self.students: if student_name == student.name: return True
return False
def add_mark(self, student_name, subject, mark): for student in self.students:
if student_name == student.name:
return student.add_mark(subject, mark) return "Student not found"

students = Students(student_data)
print(students.students)
while True:

print("Welcome to the Raspberry Pi student database") print("What can I help
you with?")
print("Enter 1 to view all report cards")
print("Enter 2 to view the report card for a student") print("Enter 3 to add a
student")
print("Enter 4 to add a mark to a student") print("Enter 5 to exit")

try:
user_choice = int(input("Choice: "))
except ValueError:
print("That's not a number I recognise") user_choice = 0

if user_choice == 1:
students.print_report_cards()
elif user_choice == 2:
enter_student = input("Which student? ")
students.print_report_cards(enter_student)
elif user_choice == 3:
```

```
enter_student = input("Student name? ")
print(students.add_student(enter_student))
elif user_choice ==4:
enter_student = input("Student name? ")
enter_subject = input("Subject? ")
num_error = True
while num_error:
num_error = False
try:
enter_mark = int(input("Mark? "))
except ValueError:
print("I don't recognise that as a number") num_error = True
print(students.add_mark(enter_student, enter_subject, enter_mark))
elif user_choice == 5:
break
else:
print("Unknown choice")
input("Press enter to continue")

print("Goodbye and thank you for using the Raspberry" , "Pi Student database")
```

Getting Extra Features from Modules

You’ve seen import lines quite a few times by now, but we haven’t really explained what they do. Actually, it’s incredibly simple—import just brings Python code from another file into the current program. If you create a file called module_example.py that contains the following:

```
print("Hello World")
```

Save it in your home directory (that is, /home/pi for the default user). Now, you enter a Python interpreter session in IDLE 3 and enter the following:

```
>>> import module_example Hello World
```

Of course, this is a fairly pointless module. Usually they contain functions or objects that can then be used. Change module_example to:

```
def hello():
print ("Hello World")
```

You’ll have to restart IDLE for it to pick this up. Once you’ve done so, you can run:

```
>>> import module_example >>> module_example.hello()
```

The line import module_example brings all the functions and classes into your project and you can access them by prefixing them with the module name. Sometimes, though, you'll only want some of the module. You can import individual parts like this:

```
>>> from module_example import hello
```

Now you can just enter the following to run the function:

```
>>> hello()
```

Notice that you don't need to prefix it with the module name. This is because it's imported into the current namespace. When you do this, you need to make sure that it doesn't clash with any of the other functions or classes you're using. You can even bring everything from a particular module into the current namespace to make it easier to use:

```
>>> from module_example import *
```

With this, you obviously have to be very careful to avoid namespace clashes.

There are a number of advantages to creating modules rather than just putting everything in the same file. It means you can reuse code between projects (remember what we've said about code reuse?). It also means that big projects don't just live in one massive file that's hard to work with. You can also split the various modules up between different members of a group to make it easier to work as a team.

In the remaining chapters, much of what we'll do will revolve round particular modules that help you add really cool features to your Raspberry Pi projects.

Summary

We haven't covered absolutely everything that's in Python. To do that would require a much larger book, and we wouldn't have space to include all the fun stuff we're going to do in the next few chapters. However, we've shown you enough to get started, and hopefully enough to understand most Python programs you see. If you get stuck at any point, just flick back to this chapter, or take a look at the Python documentation (which really does cover everything, but can be a little hard to read). You should now know:

- Variables are places you can store values.

■ Values have data types such as int, float, or bool.
■ You can group values in lists, tuples, dictionaries, or sets to make them easier to access. ■ while loops keep repeating until their condition is False.
■ for loops operate on every item in a collection of data.
■ Functions help you reuse code so you don't have to repeat it.
■ Classes allow you to encapsulate data and methods to make them easier to use.
■ There are hundreds of modules that you can import to add extra features to your project.

■ You can also create your own modules. This helps split your programs up into manageable files.

Solutions to Exercises

Exercise 1

The variables are

■ prompt_text holds"Enter a number: " which is a string.
■ user_in holds whatever the user types and is a string.

■ user_num holds the number version of whatever the user types, converted into an int.
■ i holds the numbers 1 to 9 as int.
■ even is a bool that holdsTrue orFalse (depending on whether the user's number is even).

Exercise 2

```
import random
secret = int(random.uniform(0,10))
print("I'm thinking of a number between zero and ten." , "Can you guess what it is?")
guess = 11
while guess != secret: try:
guess = int(input("Take a guess: ")) except ValueError:
print("A number! Guess a number!")
print("Well done!")
```

Exercise 3

Technically they could, but it would make the code needlessly complex because you can't change the values. Therefore, you'd have to create new tuples each

time rather than simply adding to or amending the current one.

Chapter 4
Graphical Programming

IN THE LAST chapter, we dealt a lot with how to handle data, and how to process it. After all, manipulating data is the fundamental job of a computer. You saw how to build a simple text-driven menu to help control the program. However, such interfaces went out of style in the 1980s. They still have a use in some applications, but these days, most people want to use a mouse (or touch screen).

There are modules using three different graphical toolkits that you're likely to come across— Tk, GTK, and Qt. Tk is quite an old fashioned library that is still used, but lacks some modern features. GTK is a popular toolkit, and the one that LXDE (the default Raspbian desktop) is built in. Qt (sometimes pronounced cute) is a toolkit that was originally developed by Nokia for their ill-fated smartphones. Nokia has since sold it to Digia who continues to develop it. Both GTK and Qt are free to use, and to be honest, there's not much to choose between them. This chapter uses Qt because it's a bit more portable and it's a bit better supported.

You'll need to install the pyside module before you can start using it. In LXTerminal, enter the following:
sudo apt-get install python3-pyside This may take a little while, so you might want to get a cup of tea.

Graphical User Interface (GUI) Programming

Throughout this book you're going to learn that you can create things in Python very easily if you let the right modules take care of the hard work. Graphical User Interface (GUI) programming is no different. All manner of useful widgets are available; all you have to do is pick which ones you want and add them to your project.

You can also use inheritance. In the last chapter, we introduced classes, and showed that you can create a new class that inherits all the features of a superclass. Here, you'll see how to use this to quickly create new classes that build upon the old ones.

Let’s get straight into an example. In Chapter 2 you saw the turtle module, and even how to set it to listen for keypresses. This is a little better than basic text entry, but not by much, so in the first example here, you’ll see how to create a simple GUI to control the turtle. Start with the following code (either enter it by hand, or find it in file chapter4-turtlestart.py on the website):

```
import turtle
import sys
from PySide.QtCore import * from PySide.QtGui import *

class TurtleControl(QWidget):
def __init__(self, turtle):
super(TurtleControl, self).__init__() self.turtle = turtle

self.left_btn = QPushButton("Left", self)
self.right_btn = QPushButton("Right", self) self.move_btn =
QPushButton("Move", self) self.distance_spin = QSpinBox()

self.controlsLayout = QGridLayout()
self.controlsLayout.addWidget(self.left_btn, 0, 0)
self.controlsLayout.addWidget(self.right_btn, 0, 1)
self.controlsLayout.addWidget(self.distance_spin,1 , 0)
self.controlsLayout.addWidget(self.move_btn,1 , 1)
self.setLayout(self.controlsLayout)

self.distance_spin.setRange(0, 100) self.distance_spin.setSingleStep(5)
self.distance_spin.setValue(20) #set up turtle
window = turtle.Screen() babbage = turtle.Turtle()

# Create a Qt application
app = QApplication(sys.argv)
control_window = TurtleControl(babbage) control_window.show()

# Enter Qt application main loop app.exec_()
sys.exit()
```

You can run it now, but none of the buttons will do anything (you’ll add that in a bit). First of all, let’s take a look at what’s going on here. The main part of the code is in the class TurtleControl, which inherits from Qwidget (most of the Qt classes start with the letter Q). By extending from this class, you get all the basic

functionality you need. All that you have to do is change it from a generic widget into one that fits the specific needs of this program. In short, you just have to tell it what items you want where.

There are three buttons and a spinbox (allows you to enter a number and raise and lower it— take a look at the running program to see how a spinbox works).

In addition to the items that the user will see, there's also a layout that you add these to. Qt has a few different layouts (you'll see another one later), but this program uses the QGridLayout. The grid layout is great for simple control panels like this one. It works on the basis of dividing the window up into a grid, and you tell Qt where you want the item to go in the grid. If the user resizes the window, Qt dynamically resizes the grid to take advantage of the extra space, but still keeping everything in the right portion of the grid.

To display any of the widgets on in the window, you have to add them to the layout. These are the lines like this:
self.controlsLayout.addWidget(self.right_btn, 0, 1)

The 0 and 1 are the horizontal and vertical coordinates taken from the top-left corner (that is, upside down when compared to graph coordinates). This button, then, is on the top line, one column across from the left side.

When everything's added to the layout, you need to tell the window to use that layout. This is done with the line:
self.setLayout(self.controlsLayout)
There are also some settings for widgets that you can change to alter their behavior. In this case, the spinbox is adjusted with the following:

self.distance_spin.setRange(0, 100) self.distance_spin.setSingleStep(5)
self.distance_spin.setValue(20)

This sets the minimum and maximum values, the amount each click moves it, and the initial value.
Hopefully, you'll recognise the turtle code from before. The last five lines just create the control window and execute it.

Adding Controls

All this code has created a nice looking control window, but it doesn't actually do anything. The next stage, then, is to tell Qt what you want the controls to do. This is done by connecting an event with an action. The events here will be button clicks, and actions will be the methods you want to run when that event happens.

To set this up, add the following code to the end of the _ _init_ _ method of the TurtleControl class:

```
self.move_btn.clicked.connect(self.move_turtle)
self.right_btn.clicked.connect(self.turn_turtle_right)
self.left_btn.clicked.connect(self.turn_turtle_left)

def turn_turtle_left(self): self.turtle.left(45)
def turn_turtle_right(self): self.turtle.right(45)
def move_turtle(self):
self.turtle.forward(self.distance_spin.value())
```

You'll notice that in each of the connect calls, the method in the parameter doesn't have any brackets after it like methods normally do. That is, it's this:

```
self.move_btn.clicked.connect(self.move_turtle)
```

Rather than this:

```
self.move_btn.clicked.connect(self.move_turtle())
```

This is because when you put the brackets after it, you're telling Python to run the method and send the result as a parameter, as so:

```
def move_turtle(self):
self.turtle.forward(self.distance_spin.value())
```

However, when you don't put the brackets after the method, you're telling Python to send the method itself as the parameter. That's what you need to do here so Qt knows what to run when the event happens.

You'll find the complete code on the website as chapter4-turtle.py. You can see it running in Figure 4-1.

Figure 4-1: A mouse-powered interface to a turtle.

That's more or less all the basics of PySide and Qt. It doesn't get much more complex, but there are a huge number of widgets. We won't be able to demonstrate them all, but in the next example, we'll try to show you a large enough range that you get a feel for the toolkit and you should then be able to use the other widgets as you need them.

Test Your Knowledge

Exercise 1

Extend the turtle controller program so that you can change the colour of the turtle as well as move it. We'll give you a hint to get you started. If you change the set up turtle lines to:

```
#set up turtle
window = turtle.Screen()
```

babbage = turtle.Turtle()
window.colormode(255)

Then you'll be able to set the turtle's colour with red, green, and blue values between 1 and 255, such as:
turtle.color(100,0,0)
Another thing you may find useful are QLablels. They let you add pieces of text to the window, and are created like this:
self.red_label = QLabel("Red", self) They might be useful for labeling spinboxes (nudge, nudge, wink, wink).

Creating a Web Browser

In the previous example, you saw how easy it was to link things together to create an interface. In this example, we'll use Qt's widgets to build our own web browser (which is really just a set of widgets linked together). You'll see that you don't really need any programming at all; it'll just be linking together different parts of Qt.

First of all you need to create a window for the browser. In the previous example, you created a widget that Qt put in a window for you. That's fine for simple tools, but as you build more powerful applications, it can help to create the window explicitly and add everything to that. By starting with a QMainWindow, you can add things like menus. However, that's jumping ahead, and there's quite a bit to add before menus.
The first and most important part of any web browser is the bit that actually shows the web page. You'll learn a bit more about what's actually involved in web pages in Chapter 7, but for the purposes of this chapter, all you need to know is that the QWebView widget can take care of all that for you.

In the previous example, you used a grid layout. This works well for adding a lot of controls to a window, but in this application, you're going to use a box layout. This is a little different. It's created using two different layouts: QVBoxLayout and QHBoxLayout. These are vertical and horizontal boxes, respectively. As you add items to one of these box layouts, they are placed next to each other horizontally or vertically. To create complex layouts, you just need to nest these layouts inside each other in the appropriate way. It can take a little while to get used to this way of laying out widgets on windows, but once you become familiar with it, it's incredibly powerful.

The web browser will have a typical web browser layout. That is, a bar of controls along the top, then most of the window will be taken up with the web page you're viewing. To create this interface, you'll need two layouts—a QHBoxLayout for the controls, and then a QVBoxLayout that'll take both the previous layout box and the QWebView widget. As you resize the window, Qt will adjust the layouts so that the widgets always make the best use of the space.

Hopefully, this will all become clear as you create the browser, so let's get started! The following code creates the window, and adds the appropriate layouts (the file is on the website as chapter4-web-browser-begin.py).

```
import sys
from PySide.QtCore import * from PySide.QtGui import * from
PySide.QtWebKit import *

class Browser(QWidget):
def __init__(self):
super(Browser, self).__init__()

self.webview = QWebView(self)
self.webview.load("http://www.google.com") self.setGeometry(0, 0, 800, 600)

self.menu_bar = QHBoxLayout()
self.main_layout = QVBoxLayout()
self.main_layout.addLayout(self.menu_bar)
self.main_layout.addWidget(self.webview) self.setLayout(self.main_layout)

class BrowserWindow(QMainWindow):
def __init__(self):
super(BrowserWindow, self).__init__() self.widget = Browser()
self.setCentralWidget(self.widget)

# Create a Qt application app = QApplication(sys.argv) window =
BrowserWindow() window.show()

# Enter Qt application main loop app.exec_()
sys.exit()
```

This is all the code you need for a really simple web browser. You can run it and it'll start with the Google home page, and you can navigate from there (see

Figure 4-2). The code should look familiar to you. The only new pieces are the QMainWindow (which will allow you a bit more control later on), the QWebView (which, as you can see, is a really easy way to add web browsing), and the box layouts.

Figure 4-2: The basics of a web browser.

The box layouts are now fully set up; all you need to do is add items to Browser'sself. menu_bar and they'll appear along the top of the screen.

The most basic controls for web browsing are back and forwards buttons. For this task, you can use QPushButtons in the same way you used them in the previous example. Update your code for the Browser class to the following by adding the lines in bold:

class Browser(QWidget):
def __init__(self):
super(Browser, self).__init__()

self.webview = QWebView(self)
self.webview.load("http://www.google.com") self.setGeometry(0, 0, 800, 600)

self.back_btn = QPushButton("<", self)
self.back_btn.clicked.connect(self.webview.back)
self.back_btn.setMaximumSize(20,20)

```
self.forward_btn = QPushButton(">", self)
self.forward_btn.clicked.connect(self.webview.forward)
self.forward_btn.setMaximumSize(20,20)

self.menu_bar = QHBoxLayout()
self.menu_bar.addWidget(self.back_btn)
self.menu_bar.addWidget(self.forward_btn) self.main_layout =
QVBoxLayout()
self.main_layout.addLayout(self.menu_bar)
self.main_layout.addWidget(self.webview)

self.setLayout(self.main_layout)
```

You can now run the code, and you'll have a browser with a history that you can move back and forwards through. Here again, the QWebView did all of the hard work. It only required connecting the button clicks to the QWebView'sforward and back methods.

NOTE

We've mentioned this before, but it's worth mentioning again—when you're programming, there's no point in implementing new features when you can get the functionality from a module. A little time spent learning about a module can save you a lot of time later on.

Unlike the grid layout that you used previously, in box layout, Qt has more freedom to work out what size to draw particular widgets. Sometimes this is a good thing, but other times, you need to give it a bit of guidance. In a web browser, you want the buttons to take up as little space as possible, giving all the free screen area to the web page. To do this, we call the setMaximumSize() method on the widgets we add. In the case of the buttons, we make sure they get no bigger than 20×20.

The next feature of the web browser will be a text input where the users can type the address of a site they want to visit. There are a few different Qt widgets for text entry. The most common is QTextEdit. This allows the users to display and edit text. Actually, it does more than just straight text, and it can handle images, tables, headings, and other such things.

QPlainTextEdit is another common widget that works like QTextEdit except that

it’s designed for just plain text rather than rich text. Both of these are really powerful options that you’ll probably use at some point in your programming career. However, they’re a bit too much for an address bar since they’re designed for multi-line text entry. For a single line of plain text entry (like a URL field), a QLineEdit is the best option.

You’ll also need a Go button to tell the browser to load the page. To do all this, update the Browser class to the following (updates are in bold):

```
class Browser(QWidget):
def _ _init_ _(self):
super(Browser, self).__init__()

self.webview = QWebView(self)
self.webview.load("http://www.google.com") self.setGeometry(0, 0, 800, 600)

self.back_btn = QPushButton("<", self)
self.back_btn.clicked.connect(self.webview.back)
self.back_btn.setMaximumSize(20,20)

self.forward_btn = QPushButton(">", self)
self.forward_btn.clicked.connect(self.webview.forward)
self.forward_btn.setMaximumSize(20,20)

self.url_entry = QLineEdit(self) self.url_entry.setMinimumSize(200,20)
self.url_entry.setMaximumSize(300,20)

self.go_btn = QPushButton("Go", self)
self.go_btn.clicked.connect(self.go_btn_clicked)
self.go_btn.setMaximumSize(20,20)

self.menu_bar = QHBoxLayout()
self.menu_bar.addWidget(self.back_btn)
self.menu_bar.addWidget(self.forward_btn)
self.menu_bar.addWidget(self.url_entry)
self.menu_bar.addWidget(self.go_btn) self.menu_bar.addStretch()
self.main_layout = QVBoxLayout()
self.main_layout.addLayout(self.menu_bar)
self.main_layout.addWidget(self.webview)

self.setLayout(self.main_layout)
```

def go_btn_clicked(self):
self.webview.load(self.url_entry.text())

There are a few new bits here. With the URL entry bar, there's a call to the method setMin imumSize(). Like setMaximumSize, this gives Qt some extra information about how you want the window laid out. Another new piece to help Qt lay out the window properly is the addStretch()method call. This adds a pseudo-widget that just changes shape to fill up space. In this case, it takes up all the spare room on the right side of the menu bar so Qt pushes all the controls to the left.

You can run it now and try it out. The only thing to note is that it does need you to enter http:// at the start of the web address. (The technical reason for this is because a URL or Universal Resource Locater requires this as it specifies the protocol. For example, http:// yoursite.com/document could point to something different than ftp://yoursite. com/document. Most modern browsers allow you to omit this and just assume you mean http. However, when a module asks for a URL, it usually needs the protocol prefix.)

In this case, we've added a new method to Browser called go_btn_clicked() because this gives us a little more power than just connecting methods to events. In this case, it allows you to add a parameter to the call to webview'sload method with the parameter self.url_entry.text(), which just returns the text that the user typed. At this point, you have what could realistically be called a web browser. There's nothing essential missing, although it's less powerful than mainstream browsers like Firefox, Chrome, or Midori. The next feature we decided to add is a bookmarks picker. We chose this partially because it's a useful feature, and partially because it gives us an excuse to show off another useful Qt widget, the QComboBox.

Combo box is an odd name for something you're almost certainly familiar with. They're boxes with a drop-down arrow that opens a set of choices that users can pick from. If that doesn't seem familiar now, it will be as soon as you see it.

Later on in the book we'll look at some ways you can store information between sessions, but to keep things simple, we won't let the user change or add to the bookmarks. After all, this is a chapter on user interfaces, and we want to stick to that topic.

Add the bold sections of the following to theBrowser class (the non-bold sections will let you know where to add it):

self.go_btn = QPushButton("Go", self)
self.go_btn.clicked.connect(self.go_btn_clicked)
self.go_btn.setMaximumSize(20,20)

self.favourites = QComboBox(self)
self.favourites.addItems(["http://www.google.com",
"http://www.raspberrypi.org",
"http://docs.python.org/3/"])
self.favourites.activated.connect(self.favourite_selected)
self.favourites.setMinimumSize(200,20)
self.favourites.setMaximumSize(300,20)

self.menu_bar = QHBoxLayout()
self.menu_bar.addWidget(self.back_btn)
self.menu_bar.addWidget(self.forward_btn)
self.menu_bar.addWidget(self.url_entry) self.menu_bar.addWidget(self.go_btn)
self.menu_bar.addStretch()
self.menu_bar.addWidget(self.favourites) self.main_layout = QVBoxLayout()
self.main_layout.addLayout(self.menu_bar)
self.main_layout.addWidget(self.webview)

self.setLayout(self.main_layout)
def go_btn_clicked(self): self.webview.load(self.url_entry.text())
def favourite_selected(self):
self.webview.load(self.favourites.currentText())

This is all quite straightforward, and if you run the code, you'll see a QComboBox in action.
As with the URL entry, we just call self.webview.load, but this time with a parameter that grabs the currently selected text from the combo box.

There are only two controls left to add to the menu bar, so let's make them in one edit. The first is a search bar that lets users enter a search term and then press a button to run a Google search. The second is a zoom slider bar that lets the user zoom in and out of the page.

Update theBrowser class with the bold text from the following:

self.favourites = QComboBox(self)
self.favourites.addItems(["http://www.google.com",
"http://www.raspberrypi.org",
"http://docs.python.org/3/"])
self.favourites.activated.connect(self.favourite_selected)
self.favourites.setMinimumSize(200,20)
self.favourites.setMaximumSize(300,20)

self.search_box = QLineEdit(self) self.search_box.setMinimumSize(200,20)
self.search_box.setMaximumSize(300,20)

self.search_btn = QPushButton("Search", self)
self.search_btn.clicked.connect(self.search_btn_clicked)
self.search_btn.setMaximumSize(50,20)

self.zoom_slider = QSlider(Qt.Orientation(1),self)
self.zoom_slider.setRange(2, 50)
self.zoom_slider.setValue(10)
self.zoom_slider.valueChanged.connect(self.zoom_changed)

self.zoom_label = QLabel("Zoom:")
self.webview.loadStarted.connect(self.page_loading)

self.menu_bar = QHBoxLayout()
self.menu_bar.addWidget(self.back_btn)
self.menu_bar.addWidget(self.forward_btn)
self.menu_bar.addWidget(self.url_entry) self.menu_bar.addWidget(self.go_btn)
self.menu_bar.addStretch()
self.menu_bar.addWidget(self.favourites) **self.menu_bar.addStretch()**
self.menu_bar.addWidget(self.search_box)
self.menu_bar.addWidget(self.search_btn)
self.menu_bar.addWidget(self.zoom_label)
self.menu_bar.addWidget(self.zoom_slider) self.main_layout =
QVBoxLayout()
self.main_layout.addLayout(self.menu_bar)
self.main_layout.addWidget(self.webview)

self.setLayout(self.main_layout)

```
def go_btn_clicked(self):
self.webview.load(self.url_entry.text())
def favourite_selected(self):
self.webview.load(self.favourites.currentText())
def zoom_changed(self):
self.webview.setZoomFactor(self.zoom_slider.value()/10)
def search_btn_clicked(self):
self.webview.load("https://www.google.com/search?q=" +
self.search_box.text())
def page_loading(self):
self.url_entry.setText(self.webview.url().toString())
```

Whilst there are two controls, there are four widgets to make them happen. The search box has a QLineEntry and a QPushButton as well. Together, these work in a very similar way to the URL entry control that you added earlier, except that it adds https://www.google. com/search?q= to the start of whatever you enter. For example, if you search for Raspberries, it will go to the URL https://www.google.com/search?q=Raspberries and this tells Google to search for Raspberries. This has https:// at the start rather than http://. The s stands for secure, and if you use https:// then any data between your browser and the website is encrypted. However, not every website supports https. QWebView allows you to use either protocol as long as the server supports it.

The zoom slider is a QSlider, which is another type of control that you're probably familiar with. It takes a little more setting up, though, which is what the following lines do:

```
self.zoom_slider.setRange(2, 50) self.zoom_slider.setValue(10)
```

The first sets the maximum and minimum values for the slider, and the second sets the initial value.

You connected the valueChanged action to thezoom_changed() method. Once again, this just links into one of QWebView's methods and lets it do all the hard work. The only thing zoom_changed() does is divide the value of the slider by 10 to make the zoom a bit more manageable.

If you were looking closely, you'll have noticed that this actually does a little more than adding two extra controls. It also has these lines:

```
self.webview.loadStarted.connect(self.page_loading) . . .
```

```
def page_loading(self):

self.url_entry.setText(self.webview.url().toString())
```

Which will make sure the URL entry box is always updated with the address of the current page.

Adding Window Menus

The main browser layout is finished, but there's still a bit more to add before the application's done. Remember that at the start we said that we extended a QMainWindow so that we could add menus? Well now's the time to do that.

TheWindow already has a menu; all you need to do is add things to it. These menu items are similar to widgets, except they're made from QActions.
To add a file menu with an entry to close the window, change the BrowserWindow class to the following:

```
class BrowserWindow(QMainWindow):
def _ _init_ _(self):
super(BrowserWindow, self).__init__() self.widget = Browser()
self.setCentralWidget(self.widget)

self.exitAction = QAction(QIcon('exit.png'), '&Exit', self)
self.exitAction.setShortcut('Ctrl+Q')
self.exitAction.setStatusTip('Exit application')
self.exitAction.triggered.connect(self.close)

self.menu = self.menuBar()
self.fileMenu = self.menu.addMenu('&File')
self.fileMenu.addAction(self.exitAction)
```

There's one more menu entry to add, one to open a locally stored file. This is a bit different to everything in the web browser because it opens a new window. In the new window, the users will get to browse through their files and select the one they want to open. At this point, you might be thinking that it'll require quite a bit of work to create this new window, add a whole layout, and link up all the required widgets. However, this is another place where we can just let Qt do all the hard work for us. There are a range of Qt widgets known as dialogs. These

are simple windows to perform common functions, and they can make your life a lot easier. To add an open file dialog to the web browser, update the following BrowserWindow class:

```
self.exitAction = QAction(QIcon('exit.png'), '&Exit', self)
self.exitAction.setShortcut('Ctrl+Q')
self.exitAction.setStatusTip('Exit application')
self.exitAction.triggered.connect(self.close)

self.openFile = QAction(QIcon('open.png'), 'Open', self)
self.openFile.setShortcut('Ctrl+O')
self.openFile.setStatusTip('Open new File')
self.openFile.triggered.connect(self.showDialog)

self.menu = self.menuBar()
self.fileMenu = self.menu.addMenu('&File')
self.fileMenu.addAction(self.openFile)
self.fileMenu.addAction(self.exitAction)

def showDialog(self):
fname, _ = QFileDialog.getOpenFileName(self, 'Open file', '/home')
self.widget.webview.load("file:///" + fname)
```

In this case, we don’t have to create a new object, instead we can just call the getOpenFile Name() method fromQFileDialog. This will open a new window with the title "Open File in the directory /home" (see Figure 4-3). Once the users pick the file they want, it will return two things: the filename and the filter. However, since the web browser doesn’t need to know the filter, assigning it to _ just drops it.

Figure 4-3: Opening a local file using QFileDialog.

TheQWebView can open local files using the protocol file:/// (note the three slashes), so you just need to prefix this on the filename before you can use it.

That brings us to the end of the web browser. If you haven't been following along, the complete code is on the website as chapter4-web-browser-complete.py. As with every application, there's still plenty we could add, but we've shown enough to introduce the pyside module and the Qt toolkit. It's a huge toolkit, so out of necessity, we've only been able to show you the basics. Everything is documented at http://srinikom.github. io/pyside-docs/.

Test Your Knowledge

Exercise 2

As you've seen, dialogs are great ways to add functionality to your programs quickly. In this exercise, go back to the turtle program and add a button that launches a QColorDialog, which sets the colour of the turtle.

Here are a few hints to help you out. QColorDialog.getColor() will return a value with the typeQColor. To get the RGB values out of a variable that holds a

QColor, use variable_name.getRgb()[:3]. You need the [:3] at the end because it returns a tuple with four values (the final one being the transparency, which you don't need in this case).

Summary

After consuming this chapter, you should know the following:
■ PySide is a Python library that helps you write graphical user interfaces using the Qt toolkit.
■ Qt contains a wide range of widgets that you can add to your projects to quickly create powerful interfaces.
■ You can connect actions such as button presses, combo box changes, or slider movements to method calls.

■ There are several ways you can lay out your Qt windows, including grid and box. ■ If you build your interface on a class that extends QMainWindow, you can add menus.

■ Qt includes a range of dialogs that are ready-made windows that you can add to your project.

Solutions to Exercises

Exercise 1
import turtle
import sys
from PySide.QtCore import * from PySide.QtGui import *

class TurtleControl(QWidget):
def __init__(self, turtle):
super(TurtleControl, self).__init__() self.turtle = turtle

self.left_btn = QPushButton("Left", self) self.right_btn = QPushButton("Right", self) self.move_btn = QPushButton("Move", self) self.distance_spin = QSpinBox()
self.red_spin = QSpinBox()
self.green_spin = QSpinBox()
self.blue_spin = QSpinBox()
self.red_label = QLabel("Red", self)

```
self.green_label = QLabel("Green", self) self.blue_label = QLabel("Blue", self)
self.colour_btn = QPushButton("Colour", self) self.controlsLayout =
QGridLayout()
self.controlsLayout.addWidget(self.left_btn, 0, 0)
self.controlsLayout.addWidget(self.right_btn, 0, 1)
self.controlsLayout.addWidget(self.distance_spin, 1 , 0)
self.controlsLayout.addWidget(self.move_btn, 1 , 1)
self.controlsLayout.addWidget(self.red_spin, 2,1)
self.controlsLayout.addWidget(self.green_spin, 3,1)
self.controlsLayout.addWidget(self.blue_spin, 4,1)
self.controlsLayout.addWidget(self.red_label, 2,0)
self.controlsLayout.addWidget(self.green_label, 3,0)
self.controlsLayout.addWidget(self.blue_label, 4,0)
self.controlsLayout.addWidget(self.colour_btn, 5,0)
self.setLayout(self.controlsLayout)

self.distance_spin.setRange(0, 100) self.distance_spin.setSingleStep(5)
self.distance_spin.setValue(20)

for spinner in [self.red_spin, self.green_spin, self.blue_spin]:
spinner.setRange(0, 255)
spinner.setSingleStep(5)
spinner.setValue(150)

self.move_btn.clicked.connect(self.move_turtle)
self.right_btn.clicked.connect(self.turn_turtle_right)
self.left_btn.clicked.connect(self.turn_turtle_left)
self.colour_btn.clicked.connect(self.colour_turtle)

def turn_turtle_left(self): self.turtle.left(45)
def turn_turtle_right(self): self.turtle.right(45)
def move_turtle(self):
self.turtle.forward(self.distance_spin.value())

def colour_turtle(self):
self.turtle.color(self.red_spin.value(), self.green_spin.value(),
self.blue_spin.value()) #set up turtle
window = turtle.Screen() babbage = turtle.Turtle() window.colormode(255)
```

```
# Create a Qt application
app = QApplication(sys.argv)
control_window = TurtleControl(babbage) control_window.show()

# Enter Qt application main loop app.exec_()
sys.exit()
```

Exercise 2
The following function will need to be connected to the clicked action of a button:

```
def colour_turtle(self):
self.colour = QColorDialog.getColor() self.turtle.color(self.colour.getRgb()[:3])
```

Chapter 5
Creating Games

IN THE LAST chapter you built graphical software using a GUI toolkit. This made it really easy to add buttons, text boxes, and all sorts of widgets to our software, but there's another sort of graphical application that doesn't use any of these things: games. We still need to draw things on the screen, but instead of check boxes and menus, we want fireballs, heroes, pits of doom, and all manner of fantastical graphics. Clearly, PySide isn't up to the task, but there is another module that will do exactly what we want, PyGame.

Raspbian comes with PyGame installed, but only for Python 2. Since we're building with Python 3, we'll need to install it. In this case, you'll have to compile the module from scratch, but this is a good chance to learn the process. First you need to install all the packages that PyGame will need, so open LXTerminal, and use apt-get (the package manager) to install the dependencies like so:

```
sudo apt-get update
sudo apt-get install libsdl-dev libsdl-image1.2-dev \ libsdl-mixer1.2-dev libsdl-
ttf2.0-dev libsmpeg-dev \
libportmidi-dev libavformat-dev libswscale-dev \
mercurial python3-dev
```

You'll notice that a lot of this code ends in -dev. These are the development files.

You need them when compiling software that uses those libraries.

The next step is to get a copy of the latest version of PyGame. Since we're using Python 3, we need to get the very latest version, so we'll grab it straight from the development platform with the following:

```
hg clone https://bitbucket.org/pygame/pygame cd pygame
```

These two lines will download the current version of PyGame into a new directory called pygame, then move into it. Once it's there, you can build and install the module for Python 3 with:

```
python3 setup.py build
sudo python3 setup.py install
```

If everything's gone well, you'll now be able to use PyGame in Python 3. To test that it works, open up the Python Shell in IDLE, and enter:

```
>>> import pygame
```

If there are any errors, then something has gone wrong and you'll need to go back and repeat the steps before continuing with the chapter. Whilst you're in the shell, you can try out a few things to see how PyGame works.

```
>>> pygame.init()
>>> window = pygame.display.set_mode((500, 500)) >>> screen =
pygame.display.get_surface()
>>> rect = pygame.Rect(100, 99, 98, 97)
>>> pygame.draw.rect(screen, (100, 100, 100), rect, 0) >>>
pygame.display.update()
```

You should see this open a new window and draw a grey rectangle. The first line just gets PyGame up and running. The second line opens a new window. The parameter - (500, 500) is a tuple containing the width and height of the new window. Notice how there's two opening and closing brackets? One set denotes the parameter and the other, the tuple. The third line gets the surface that you can draw on and stores it in the variable screen.

There are two main PyGame classes that you'll be using: Rect and Sprite. The second one we'll look at later, but the first, Rect, is absolutely critical to the way PyGame works. In the fourth line, you create one of these rectangles, which has its top-left corner at coordinates 100, 99, and is 98 pixels wide by 97 tall. The

one thing you need to know about PyGame coordinates is that they start from the top-left corner of the screen, so compared to normal graph coordinates, they're upside down.

Rectangles aren't always displayed on the screen (they serve a number of other useful purposes as you'll discover later), but this one will be drawn, and that's done in the next line. The parameter (100, 100, 100) holds the red, green, and blue colours (each one is from 0 to 255), and the 0 is the line thickness (0 means fill, 1 or higher means line thickness). If you were paying attention as you typed, you'd notice that the rectangle doesn't appear in the window. That's because you need to update the screen for any changes to take effect. This we do in the final line.

Try drawing a few other rectangles on the screen to get a feel for how all the different parameters affect the shape.

Building a Game

Those are the very basics of PyGame. Now onto the game you'll build, which is a simple platform game where you control a character who has to run and jump through a level to try and reach a goal. To make it a little tricky, we'll rain down fireballs that she has to dodge as she makes her way there, and if she falls off the platform, there'll be a burning pit of doom waiting to finish her off.

When programming, we don't usually start with the first line. Instead, we make a plan of how we think the program will work. This is like the skeleton of the program that we'll then flesh out. We do this by designing all the classes and their methods, but leaving the implementation blank. As we program, we add flesh to this design until, hopefully, we end up with a finished program. For our game, we have the following design (you can type it out, but it'll be easier to download it from the website as chapter5-pygame-skell.py and add to it as you go through the chapter):

```
import pygame
import sys
from pygame.locals import * from random import randint

class Player(pygame.sprite.Sprite):
'''The class that holds the main player, and controls how they jump. nb. The
```

```
player doesn't move left or right, the world moves around them'''

def __init__(self, start_x, start_y, width, height): pass

def move_y(self):
'''this calculates the y-axis movement for the player in the current speed'''

def jump(self, speed):
'''This sets the player to jump, but it only can if its feet are on the floor'''
pass

class World():
'''This will hold the platforms and the goal. nb. In this game, the world moves
left and right rather

than the player'''
def __init__(self, level, block_size, colour_platform, colour_goals):
pass

def move(self, dist):
'''move the world dist pixels right (a negative dist means left)'''
pass

def collided_get_y(self, player_rect):
'''get the y value of the platform the player is currently on'''
pass

def at_goal(self, player_rect):
'''return True if the player is currently in contact with the goal. False otherwise'''
pass

def update(self, screen):
'''draw all the rectangles onto the screen''' pass

class Doom():
'''this class holds all the things that can kill the player'''
def __init__(self, fireball_num, pit_depth, colour): pass

def move(self, dist):
'''move everything right dist pixels (negative dist means left)'''
```

```
def update(self, screen):
'''move fireballs down, and draw everything on the screen'''
pass

def collided(self, player_rect):
'''check if the player is currently in contact with any of the doom.
nb. shrink the rectangle for the fireballs to make it fairer'''
pass

class Fireball(pygame.sprite.Sprite):
'''this class holds the fireballs that fall from the sky'''
def __init__(self): pass

def reset(self):
'''re-generate the fireball a random distance along the screen and give them a
random speed''' pass

def move_x(self, dist):
'''move the fireballs dist pixels to the right (negative dist means left)'''
pass

def move_y(self):
'''move the fireball the appropriate distance down the screen
nb. fireballs don't accellerate with gravity, but have a random speed. if the
fireball has reached the bottom of the screen, regenerate it'''
pass

def update(self, screen, colour):
'''draw the fireball onto the screen''' pass

#options
#initialise pygame.mixer #initialise pygame
#load level
#initialise variables
finished = False
#setup the background
while not finished:

pass
#blank screen
```

```
#check events
#check which keys are held
#move the player with gravity
#render the frame
#update the display
#check if the player is dead
#check if the player has completed the level #set the speed
```

There are only a few pieces of actual code here, but it’s enough for us to know what’s going on. This is actually a legal Python program, so you can enter it and run it. It won’t do anything except spin itself round in a loop until you hit Ctrl+C to stop it, but this gives you a base to work from. As we add pieces, we’ll make sure it stays as a running program so you can constantly check how it’s playing, and that it’s working correctly. Notice the pass statement in every method. This statement does nothing, but Python complains if you have a method with no code in it, so this simple line is required for the code to run.

This is quite a good way to start when you write your own programs. Rather than trying to create the whole thing in one go, you can start by planning how everything will work, and then build up bit-by-bit until you have a fully working program. If needed, you can always change your plan, but it helps to know what you’re working towards.

Initialising PyGame

We’ll now add a couple of things to get it started. Add the following code to the file where the comments match.

```
#options
screen_x = 600
screen_y = 400
game_name = Awesome Raspberry Pi Platformer"

#initialise pygame
pygame.init()
window = pygame.display.set_mode((screen_x, screen_y))
pygame.display.set_caption(game_name)
screen = pygame.display.get_surface()
#initialise variables
```

```
clock = pygame.time.Clock()

#check events
for event in pygame.event.get(): if event.type == QUIT: finished = True
#set the speed clock.tick(20)
```

You should recognise the section under initialise pygame from earlier. The only change here is that we've taken the screen size out and stored it in two variables. This is so that you can easily change it later without having to try to remember what code does what. All the key options will be stored as global variables in the same place to allow you to tweak the way the game works.

The section under #check events just waits until the user clicks on the cross to close the window, then exits the loop. The two lines for the clock use PyGame's timer to moderate the speed the loop runs at. Since each turn of the loop will correspond to a singe frame of the game, we need to make sure it doesn't run too fast (otherwise, all the action would be over before the user had a chance to do anything). You could just tell Python to sleep for a certain amount of time like you did in the simple turtle game in Chapter 2, but this has the slight problem that you don't know how long the rest of the loop will take. If you run it on a slower computer, the game will run at a different speed to on a fast machine. clock.tick(fps) is the solution to this. It tries to hold the loop atfps loops per second by pausing the loop for the appropriate amount of time, taking into account how long the rest of the loop has taken to run. In the previous code, the final line will calculate how long to wait for so that the loop runs exactly 20 times a second.

Now, let's start building the classes, starting with the Player class. Add the following initialisation method to it:

```
def __init__(self, start_x, start_y, width, height):
pygame.sprite.Sprite._ _init_ _(self)
self.image = pygame.transform.scale(
pygame.image.load(player_image), (width, height))

self.rect = self.image.get_rect()
self.rect.x = start_x
self.rect.y = start_y
self.speed_y = 0
self.base = pygame.Rect(start_x, start_y + height, width,2)
```

The key thing about Player is that it inherits from pygame.sprite.Sprite. This allows you to use it to draw an image, but first you have to set up two key variables: self.image and self.rect. Once these are set up, the parts it inherits from pygame.sprite.Sprite will allow you to draw it. Fairly obviously, self.image is the image that you want the sprite to have and self.rect is the rectangle that PyGame will draw it in. Once set up, you can then move and manipulate this rectangle just like any other, and PyGame will move the image round the screen for you. As you'll see later, you move rectangles round by updating their x and y attributes.

The penultimate local variable (speed_y) is used to keep track of the player's up and down speed as she jumps, whilst the final one (base) is a very short rectangle that represents the character's feet. You'll use this to check whether she's standing on a platform.

The player is now ready to draw on the screen, but first you'll need a bit more code. Add the following to the appropriate areas:

```
#options
player_spawn_x = 50
player_spawn_y = 200
player_image = "lidia.png"

#initialise variables
player = Player(player_spawn_x, player_spawn_y, 20, 30) player_plain =
pygame.sprite.RenderPlain(player)

#render the frame
player_plain.draw(screen)
#update the display pygame.display.update()
```

In order to draw a sprite, you need to give it an image to draw. If you're artistic, you may want to create this yourself. However, there's a great collection of images you can use for games at http:// opengameart.org. These are all free to download and use in your games, but some of them have licenses that say if you make a game with it, and you distribute that game to other people, you have to let them have the Python code so they can modify it, and build other games with your code if they want to. This concept is known as open source (see note). There are links on every file of http://opengameart.org that tell you exactly what

license they're under. The important thing to realise is that you don't have to worry about it unless you distribute your game to other people. Not all of the files there will work well with PyGame. We recommend sticking with .png files for images. After a bit of searching, we like using http://opengameart.org/sites/default/ files/styles/medium/public/lidia.png for our game's main character, but feel free to pick a different one (although you'll have to update the player_image variable). As the code is currently, it'll look for the player_image image file in the same directory it's being run from.

Open Source

Open source is a concept in which people share the programs they've written. Not just the executable files that can be run, but also the actual program code so that other people can modify it in whatever way they see fit. For example, the Linux operating system on the Raspberry Pi is open source, so are all of the tools that come standard, like the Midori web browser and even Python itself. Roughly speaking, there are two different types of open source—ones where you have to share any modifications you make, and ones where you don't. Creative Commons is a similar concept to open source, instead for works of art like pictures, sounds, and writing. When incorporating pieces of open source software or Creative Commons artwork into your programs, it's important that you understand what your responsibilities are. There will always be a link to the exact license that it's released under. In the case of Creative Commons, these are easy to read and understand. If you need to share your work in order to comply with a license (or if you want to make your work open source), the easiest way is to put it on an open source hosting website such as github.com, which will host it for free.

To draw a sprite, you also need to add it to a RenderPlain. This just creates the object that you draw on the screen. Here, the player object has its own RenderPlain object called player_plain.

You should now be able to run the code and it'll display the sprite at the coordinates player_spawn_x, player_spawn_y. Technically, it'll be redrawing it 20 times a second, but since it's always in the same place, you can't tell. It should look like Figure 5-1.

Figure 5-1: The simple beginnings of the platform game.

With the character drawn, the next task is to make her move. This is just some test code to make sure the animation is working properly. A little later you'll update this to let your character jump. Add the following method to the Player class:

```
def move_y(self):
self.rect.y = self.rect.y + 1
```

This will just move the character slowly down. For this to do anything, though, we'll have to add the following to the appropriate places in our game loop:

```
#blank screen
screen.fill((0, 0, 0))
#move the player with gravity player.move_y()
```

You can now run the code, and you'll see the player move down the window until she disappears off the bottom. We can't do much more with her until we've built a world for her to move round in.

Creating a World

We want to make it as easy as possible to extend this game and make it awesome, so we want it to be really easy to design new levels. We've done this by defining each level as a list of strings. Each string corresponds to a line on the screen, and each character in the string corresponds to a block on that line. A - means that there's a platform there, a G means there's a goal there (the place the player has to reach to finish the level), and anything else means it's blank. To create the basic level, then, add the following under #options:

```
level=[
" ", " ", " ", " ", " ", " ", " ", " --- G", " -- -- --- ------", " -- - ------- "]]

platform_colour = (100, 100, 100) goal_colour = (0, 0, 255)
```

If you've entered it correctly, all the lines should be the same length. This is a really simple level, but it'll do for testing. The final two lines also set the colours for the platform and the goal, respectively. As we saw with the rectangle at the start of this chapter, these colours are in RGB values.

Now add the following code to the __init__ method of the World class to load this:

```
def __init__(self, level, block_size, colour_platform,
colour_goals):

self.platforms = []
self.goals = []
self.posn_y = 0
self.colour = colour_platform
self.colour_goals = colour_goals
self.block_size = block_size

for line in level:
self.posn_x = 0
for block in line:

if block == "-":
self.platforms.append(pygame.Rect( self.posn_x, self.posn_y,

block_size, block_size)) if block == "G":
self.goals.append(pygame.Rect( self.posn_x, self.posn_y, block_size,
```

block_size)) self.posn_x = self.posn_x + block_size self.posn_y = self.posn_y + block_size

The code is fairly simple; it just loops through every line, then every character in the line, and builds the appropriate rectangles when it finds the appropriate characters. Before you can use these blocks, you need to also add the update method to theWorld class, which will draw the blocks onto the screen:

```
def update(self, screen):
'''draw all the rectangles onto the screen'''

for block in self.platforms:
pygame.draw.rect(screen, self.colour, block, 0)
for block in self.goals:
pygame.draw.rect(screen, self.colour_goals, block, 0)
```

Now you just need to add the following code to create the objects and render them. As always, put them in the right place by looking at the comments.

```
#initialise variables
world = World(level, 30, platform_colour, goal_colour)
#render the frame world.update(screen)
```

You can now run the game, but you still won’t find much to play. The world will be drawn, but the character will slowly fall through the level, and continue falling until she disappears off the screen.

Detecting Collisions

Fortunately, it’s really easy to get two game elements to interact using PyGame’s Rect’s colliderect() method. This is incredibly simple, and the format is rect1.colliderect(rect2)

Where rect1 and rect2 are rectangles. This will return True if the two rectangles overlap, and False otherwise. You can use this to detect when the player is in contact with the world so she doesn’t just fall through it. Start with the World class and add:

```
def collided_get_y(self, player_rect):
'''get the y value of the platform the player is currently on'''
return_y = -1
for block in self.platforms:
```

```
if block.colliderect(player_rect): return_y = block.y - block.height + 1 return
return_y
```

This doesn’t just check if the player is in contact with any part of the world, but also returns the top of the rectangle that the player is touching, or -1 if the player isn’t touching anything. The next step is to update the Player class to move or not, as appropriate

```
def move_y(self):
'''this calculates the y-axis movement for the player in the current speed'''
collided_y = world.collided_get_y(self.base)

if self.speed_y <= 0 or collided_y < 0: self.rect.y = self.rect.y + self.speed_y
self.speed_y = self.speed_y + gravity

if collided_y > 0 and self.speed_y > 0: self.rect.y = collided_y
self.base.y = self.rect.y+self.rect.height
```

You’ll also need to add one thing to help the player fall realistically:

```
#options
gravity = 1
```

You can now run this. The player will now fall until she rests on top of the platform, as shown in Figure 5-2.

Figure 5-2: Our heroine can now stand atop the world we've created.

Let's take a look at move_y() to see why this happens. The code deals with two possibilities, and each one has its own if block. The first possibility is that the players are free to move up or down depending on their current speed and gravity. This is the case if they're not touching any part of the world (that is, collided_get_y returns -1). We also want the players to be able to jump up through platforms to get to higher ones, so if the players are currently moving upwards (that is, if self.speed_y <= 0), we treat the character as though she's not touching the world. If either of these conditions is true, then we run:

self.rect.y = self.rect.y + self.speed_y self.speed_y = self.speed_y + gravity

This moves the character by a distance determined by the current speed, then updates the current speed by gravity (gravity is an acceleration, so this mimics the physics of the real world).

The second possibility (which is checked by the second if block) is that the players are currently in contact with a platform and they're not moving up. If this is the case, the program just adjusts the coordinates of the character so that she's correctly aligned with the world. This is because if the players fall at more

than one pixel per frame, it's possible that they could be several pixels below the top of the platform before this is checked. Later on, you may notice that sometimes you can see the character going down then back up slightly as she lands from higher falls, but this mimics people recovering from a landing.

Moving Left and Right

You've almost got something that represents a game now. There's a player and a world, but the players still can't explore it. The next thing to add is movement. There'll be two kinds of movements. Firstly we'll allow the players to jump, but only if they're in contact with the floor, and secondly they'll be able to move left and right.

First add the jump() method to thePlayer class. Since you already have the character falling, jumping is simple. All you have to do is make sure she's on the ground, then set her moving upwards and let her fall on her own:

```
def jump(self, speed):
if world.collided_get_y(self.base)>0: self.speed_y = speed
```

Now to move the player left and right. Actually, it's easier if the players stay still and the world moves left and right behind them. This gives the effect of moving without the problem of the character disappearing off the screen.

All you need to do is loop through every rectangle in the world (that is, both the platforms and the goal), and move them by a given offset. We could just update the rectangle's x and y attributes, but there are also two methods in the Rect class that you and use: move(x-distance, y-distance) and move_ip(x-distance, y-distance). move() returns a new rectangle that is the same but offset by the given distances, whilst move_ip() changes the current rectangle by the distances (ip stands for in place). Since we don't want to keep creating new rectangles every time we move, we'll use move_ip(). The code is

```
def move(self, dist):
for block in self.platforms + self.goals: block.move_ip(dist, 0)
```

The only thing left is to add code to the main loop to run the appropriate methods when keys are pressed. There are two ways of doing this in PyGame. The first way involves listening for keypress events and then taking action depending on which keypresses are detected. This is good if you only care about

when keys are pressed. The second way is using pygame.key. get_pressed() to return a list with an entry for each key. The value of the item that corresponds to a key will be True if it's held down and False if it isn't. This second method works better if you want users to be able to hold down keys to keep moving. Since we do want users to be able to hold down keys, add the following to the appropriate part of the game loop:

```
#check which keys are held
key_state = pygame.key.get_pressed() if key_state[K_LEFT]:

world.move(2)
elif key_state[K_RIGHT]:
world.move(-2)
if key_state[K_SPACE]:
player.jump(jump_speed)
```

Note that K_LEFT, K_RIGHT, and K_SPACE are all constants that we imported with pyg ame.locals. There are also K_a to K_z for the letter keys.
Add the option for the speed of the jump (negative because the PyGame coordinate system is from the top-left):

```
#options
jump_speed = -10
```

If you run the code now, you'll have what could be called the basics of a platform game (see Figure 5-3). You can move the character about, and jump over gaps. However, two crucial parts are missing. Firstly, there's no way to complete the level, and secondly, there's nothing trying to stop you.

Figure 5-3: The basics of a platform game running on a Raspberry Pi with less than two hundred lines of Python.

Reaching the Goal

Let's deal with the first of these shortcomings first. Partly because it's easier and partly because you'll then have a game you can play-test. Since the code creates, displays, and moves the goal as appropriate, all we have to do is find out if the character's at the goal. This is done in two stages. First, add the following method to the World class:

```
def at_goal(self, player_rect):
for block in self.goals:
if block.colliderect(player_rect):
return True
return False
```

This works in exactly the same way as the collide_get_y() method that we created earlier, except that it only returns True orFalse. You then need to check this method in the game loop, so add:

```
#check if the player has completed the level
```

```
if world.at_goal(player.rect):
print("Winner!")
```

If you save and run the code now, you'll find that you can run and jump to the goal, then finish the level. You can even fall off the platform and disappear into the abyss never to be seen again, but it still doesn't have much of a challenge to it.

Making a Challenge

To add some game play, there'll be a class called Doom, which holds all the things that can kill the player. In this game, there are two challenges to avoid. Firstly, there's the burning pit of doom that covers the bottom of the screen. This will kill the players if they fall into it. Secondly, and more importantly from the perspective of the game play, there'll be fireballs that drop down from the sky. The players will have to dodge these as they make their way towards the goal.

Firstly, add the burning pit of doom. We'll draw this as a rectangle along the bottom of the screen. Add the following to the Doom class:

```
def _ _init_ _(self, fireball_num, pit_depth, colour): self.base = pygame.Rect(0, screen_y-pit_depth, screen_x, pit_depth)
self.colour = colour
def collided(self, player_rect):
return self.base.colliderect(player_rect)

def update(self, screen):
'''move fireballs down, and draw everything on the screen'''
pygame.draw.rect(screen, self.colour, self.base, 0)
```

Also add the following to the appropriate parts of the options, variables, and game loop:

```
#options
doom_colour = (255, 0, 0)
#initialise variables
doom = Doom(0, 10, doom_colour)

#render the frame
doom.update(screen)
#check if the player is dead if doom.collided(player.rect):
```

```
print("You Lose!")
```

This should all be fairly self explanatory. Notice that you don’t need to move the burning pit of doom rectangle, as it should always cover the bottom of the screen. The call to the update() method is to move the fireballs, so let’s look at them now.

We’ll add the whole class in one go here:

```
class Fireball(pygame.sprite.Sprite):
'''this class holds the fireballs that fall from the sky''' def _ _init_ _(self):

pygame.sprite.Sprite.__init__(self)
self.image = pygame.transform.scale(
pygame.image.load(fireball_image),
(fireball_size, fireball_size)) self.rect = self.image.get_rect()
self.reset()

def reset(self):
self.y = 0
self.speed_y = randint(fireball_low_speed,

fireball_high_speed) self.x = randint(0,screen_x)
self.rect.topleft = self.x, self.y

def move_x(self, dist):
self.rect.move_ip(dist, 0)
if self.rect.x < -50 or self.rect.x > screen_x:

self.reset()

def move_y(self):
self.rect.move_ip(0, self.speed_y) if self.rect.y > screen_y:

self.reset()
```

As you can see, this class extends the Sprite class in the same way that the Player class does. The move_x() method works in a similar way to the equivalent method in World, except that here it has to move only a single fireball because we will have one of these fireball objects for each fireball.

To keep up the challenge, the fireballs should constantly fall from the sky. There are a few ways of achieving this, but we've chosen to create a fixed number of fireballs and simply reset them whenever they go off the screen. This reset() method places the fireballs at a random position along the top of the screen and gives them a random velocity.

The randint(a, b) method returns a random integer between a and b, inclusive (that is, including the values of a and b). The screen_x variable makes sure it's on the screen, and two global variables (fireball_low_speed and fireball_high_speed) set the range of speeds a fireball can move at. These numbers are pixels per frame. You don't need a collide method here because you'll deal with that in the Doom class.

Now, update theDoom class to:

```
class Doom():
'''this class holds all the things that can kill the player''' def __init__(self,
fireball_num, pit_depth, colour):

self.base = pygame.Rect(0, screen_y-pit_depth, screen_x, pit_depth)
self.colour = colour
self.fireballs = []
for i in range(0,fireball_num):
self.fireballs.append(Fireball())
self.fireball_plain = pygame.sprite.RenderPlain( self.fireballs)

def move(self, dist):
for fireball in self.fireballs: fireball.move_x(dist) def update(self, screen):
for fireball in self.fireballs:
fireball.move_y()
self.fireball_plain.draw(screen)
pygame.draw.rect(screen, self.colour, self.base, 0)
def collided(self, player_rect):
for fireball in self.fireballs:
if fireball.rect.colliderect(player_rect):

hit_box = fireball.rect.inflate(
-int(fireball_size/2),
-int(fireball_size/2))
```

```
if hit_box.colliderect(player_rect): return True
return self.base.colliderect(player_rect)
```

As you can see, this creates a list of fireballs and adds them to fireball_plain. This is a RenderPlain that works in the same way as player_plain, and allows you to draw the fireballs on the screen. Notice that there are global variables for the number and size of fireballs. Changing these has a dramatic effect on how the game plays, and in many ways, they're the key variables for changing difficulty.

The collided() method is a little different to the previous ones we've done so far. It compares the player to a rectangle half the size of the fireball rectangle. This is because neither the player nor the fireball are perfect rectangles, and the two bounding rectangles can collide even if the actual sprites are some distance apart. This is extremely frustrating for the person playing the game. The method we've used isn't perfect, but it errs towards the player not dying. In other words, it may be possible for the player to skim a fireball and get away with it, but if this collide method returns True then there's definitely a collision.

NOTE

It is actually possible to do perfect sprite collision in PyGame using pygame.sprite. collide_mask(sprite1, sprite2). However, this uses significantly more computing power, and is a bit overkill for this task.

With these two classes added, you just need the following code to get it all working:

```
#options
fireball_size = 30
fireball_number = 10
fireball_low_speed = 3
fireball_high_speed = 7 fireball_image = "flame.png"
```

Change the initialisation of Doom to include fireballs:
doom = Doom(fireball_number, 10, doom_colour)
You'll also need to add lines to the keypress section to make the fireballs move with the background (the lines in bold are the ones you need to add):

```
#check which keys are held
```

key_state = pygame.key.get_pressed() if key_state[K_LEFT]:

world.move(2)
doom.move(2)
elif key_state[K_RIGHT]:
world.move(-2)
doom.move(-2)

Again, we're using a sprite that we got from http://opengameart.org. In this case it's the one from http://opengameart.org/sites/default/files/flame.png. Feel free to pick another or draw your own. For this to work, the file has to be downloaded and saved in the same directory you're running the game from. Alternatively, you can give the sprites an absolute path. For example, if you're saving everything in the directory /home/ pi/my_game/, you could change the line:

fireball_image = "flame.png"
to
fireball_image = "/home/pi/my_game/flame.png"
That way it would work wherever you ran the game from. Now save and run, and the game should look like Figure 5-4.

Figure 5-4: It's a little rough round the edges, but it's a working platform game.

Making It Your Own

The mechanics of the game are now in place. Players have to move through the world, dodge the fireballs, and get to the goal. There's still a little polish left to add, but the basics are there. Now's a great time to start making it your own. After all, this isn't a chapter about how to copy code until you have a game; this is a chapter about building your own game. By now you should know enough about what the various bits do to start customising it. The options section is the best place to start.
Depending on your monitor, you may want to change the size of the window. If you think it's a bit too easy, add some more fireballs, or make them larger. Perhaps you want to jump higher, or run faster. All of these should be pretty easy. In fact, you should have learned enough in earlier chapters to now make a simple game menu that you can add to the start of the game. You could make it a simple text-based menu that goes just before #initialise pygame, and lets you set the level of difficulty. At harder levels you could … actually, we'll let you work that out for yourself. If you're feeling ambitious, you could make this menu graphical rather than text-based.

Adding Sound

Hopefully, you now have your own customised version of our game, but don't worry if you don't. The rest of this chapter will still work and you can go back and add your own tweaks later.

Now it's time to add a bit of flare. These are things that don't affect the mechanics of the game, but make it more enjoyable to play. The first is a sound effect, and the second is a background.

Before we can add sounds, we need to initialise the mixer. This basically just gets the sound infrastructure set up and ready to play. It's done with the following code:

```
#initialise pygame.mixer
pygame.mixer.pre_init(44100, -16, 8, 2048)
pygame.mixer.init()
```

This allows us to play up to eight sounds at once, although at first, we'll just add a jumping sound effect. Again, we've gone to http://opengameart.org. This time

the file is . This time the file is jump.ogg, so again you'll need to download this or a corresponding file. We'll add this file to the options with:

```
#options
jump_sound = "qubodup-cfork-ccby3-jump.ogg"
```

NOTE

MP3 sound files will sometimes work, but can be a little persnickety. They have also been known to crash PyGame games, so it's best to stick with OGG files if you can.

Then we need to update the Player class to play the noise at the appropriate time. Add the following to the end of the _ _init_ _() method:

```
self.sound = pygame.mixer.Sound(jump_sound)
```

You'll also need to change the jump() method so that it is

```
def jump(self, speed):
if world.collided_get_y(self.base)>0: self.speed_y = speed
self.sound.play()
```

That's all you need to add a bit of sound to the game. You should now find it quite easy to add more effects, like one that plays when players reach the goal, or one that plays when they die. You could also add some background music. However, remember that most music you buy is copyrighted. You can include it in a game you make for yourself without any problems, but if you want to distribute your game, you could get into trouble. Instead, take a look at http:// freemusicarchive.org. Like http://opengameart.org, this site contains a wide range of files that you can download and include in your own games. There's a wide range of styles, so you'll almost certainly find something you like. Many of these are also licensed so that if you distribute your game, you also have to distribute the source code.

Adding Scenery

The second nicety we'll add to the game is a background. Of course, you'll need a background image to do this, and we've gone for the file background.png that's in http:// opengameart.org/sites/default/files/background.zip. This gives a nice, countryside backdrop, but you could alter the feel of the game by going for something darker and moodier. You'll need to add the following to the options to bring in the file:

```
#options
background_image = "background.png"
```

Before, when using images, you extended the Sprite class to create a new class (such as Player and Fireball) to draw them. However, since you don’t need to manipulate the rectangle for this, or do any collisions, you can simply load it as an image. This is done with the following code:

```
#set up the background
background = pygame.transform.scale(pygame.image.load( background_image),
(screen_x, screen_y)).convert() bg_1_x = -100
bg_2_x = screen_x - 100
```

The first line loads the image and scales it to the screen size. It also runs convert() on it. This converts the image from a PNG to a PyGame surface. This makes it render on the screen much faster, which is especially important for an image of this size. You could have done this with the other images you’ve used, but then you’d lose the transparent sections round the images, making them purely rectangular. The second and third lines set up the variables that hold thex positions of the image. There are two of these because we’ll draw the image twice to create a constantly looping background that the players can never move off the end of.

To move the background, update the appropriate section of the game loop to:

```
#check which keys are held
key_state = pygame.key.get_pressed() if key_state[K_LEFT]:

world.move(2)
doom.move(2)
bg_1_x = bg_1_x + 1
bg_2_x = bg_2_x + 1
if bg_1_x < screen_x:
bg_1_x = -screen_x

if bg_2_x < screen_x:
bg_2_x = -screen_x
elif key_state[K_RIGHT]:
world.move(-2)
```

```
doom.move(-2)
bg_1_x = bg_1_x - 1
bg_2_x = bg_2_x - 1
if bg_1_x > -screen_x:
bg_1_x = screen_x
if bg_2_x > -screen_x:
bg_2_x = screen_x
```

There's quite a bit going on here. Firstly, did you notice that we move the background by less than we move the world or the doom? This is called parallax scrolling. It creates the appearance of depth by moving objects farther behind at different speeds. It's like when you look out of the window of a moving vehicle and the objects close to you appear to be moving faster than those farther away. It's not exactly advanced 3D graphics, but it helps create a sense of depth. If you want to take things further, you can add layers of backgrounds here. For example, you could draw some trees that move only a bit slower than the platforms, then some hills that move bit slower than the trees, and finally a sun that moves really slowly. As with the sounds, you can take this as far as you want to go.

The second thing that's going on in the code is the if blocks that move the background. Whenever the image moves so it's off one side of the screen, the program moves it back to the other side of the screen. This creates the infinitely scrolling background that constantly loops between the two background images. The only thing left to do is draw the image on the screen:

```
#render the frame
screen.blit(background,(bg_1_x, 0)) screen.blit(background,(bg_2_x, 0))
```

These have to be the first lines under #render the frame because otherwise they'll be drawn over the top of the other parts. Figure 5-5 shows the final game.

Figure 5-5: The game with all the elements.

Adding the Finishing Touches

This, in essence, is the game fully complete. However, there is one more bit we'll add to make it easier to use. So far, we've been using a level that's hard coded into the game. However, it would be much better if we could set it up so that the users can specify a file to load, and the program would pull the level out of the text in that file. Since the levels are defined by text, this should be quite easy.

Running python3 chapter5-platformer.py (or whatever you've called the file) will run the default level that's inside the main game file, but python3 chapter5-platformer. py mylevel will run the game with the level specified in the file mylevel. To do this, we need to use sys.argv. This is in the sys module, and it's a list containing all the arguments that get passed to Python. sys.argv[0] will be the name of the script we're running, so the argument that contains the filename (if it exists) will be sys.argv[1]. All we have to do, then, is add the following to our program:

```
#load level
if len(sys.argv) > 1:
```

```
with open(sys.argv[1]) as f: level = f.readlines()
```

The specified file will be read in the same way as the array we've been using up until now. That is, a - is a platform and a G is a goal, and multiple lines define the multiple levels of the game.

NOTE

If you haven't been following along, you can download the entire game from the website as chapter5-platformer.py, although we strongly encourage you to work through this chapter as it'll help you understand much more about what's going on.

Taking the Game to the Next Level

We could go on and on and add more and more to the game. However, we'll end the tutorial here. Not because the game's finished, but because you should now know enough to finish it by yourself. We're not going to dictate to you what the game should have—it's your game, add what you want. However, we will give you some ideas for how to move on:

- If you haven't already tried tweaking the options, try that now.
- Creating new levels is a great way to make the game feel like it's your own.

- The artwork we've used is only a suggestion. See what you can find online, or try making some of your own.
- Add sprites to the items that are currently just rectangles, like the platform and the burning pit of doom.
- Build up levels into worlds. Each world could have a different theme, and different artwork to match the theme.
- Add things that players can collect. These could be coins that count towards the score, or power-ups that allow the players to run faster or jump higher.

- Add more things that could kill the players. This could be, for example, something that constantly moves right so the players have to keep moving in order to avoid it, something that comes for the players and they have to kill by jumping on, or something that shoots up from below.
- Score each level on completion. You could do this with timing, or objects that the players collect, or something else.

■ Animate the sprites. Many of the images on http://opengameart.org have a range of poses to allow you to animate objects by constantly scrolling between a set of images.

■ It could speed up if you hold the arrow key down rather than just always moving at a constant speed, or there could be some Run button that enables the players to move faster.

These are just a few ideas to get you started. It’s not intended to be a complete list of everything you can do with the game, so get your creative juices flowing and see what you can come up with. If it gets good, you could submit it to the Raspberry Pi store and let other people play it. Just remember the licenses of any images or sounds you’ve used.

Realistic Game Physics

PyGame is a great module for creating simple games. As you’ve seen, it’s really easy to draw objects on the screen and move them round. However, sometimes you need a bit more power. In the previous example, the character fell as though affected by gravity, but the rest of the physics were a bit off. If you need objects that can interact with each other in more realistic ways, such as bouncing of each other, you’ll need to use a physics library.

PyMunk is one such module that allows you to create more life-like games. Using it, you can create a space and add objects, then let PyMunk work out how they’ll interact.

You can download PyMunk from http://code.google.com/p/pymunk/downloads/list (you’ll need the source release). Once it has downloaded, you can unzip it and move into the new directory with the following (use LXTerminal rather than Python to run these commands):

unzip pymunk-4.0.0.zip cd pymunk-4.0.0

Unfortunately, there is a slight error in the build file that stops it building correctly on the Raspberry Pi. In order to install it, you need to open setup.py with a text editor (such as LeafPad) and find the lines:

elif arch == 32 and platform.system() == 'Linux': compiler_preargs += ['-m32', '-

O3']]
Make sure it’s the line with arch == 32, not 64. Delete '-m32' from the second line so that it reads:
compiler_preargs += ['-O3']]
Then save the file. Now you’ll be able to install PyMunk with:

python3 setup.py build_chipmunk python3 setup.py build
python3 setup.py install

Again, these both have to be entered in LXTerminal in the PyMunk directory. This may take a while, but once it’s complete, you can check that it worked by opening Python and entering the following:
>>> import pymunk
Hopefully, you won’t get any errors.
The following example is on the website at chapter5-pymunk.py:

```
import pygame, pymunk
from pygame.locals import * from pygame.color import * from pymunk import
Vec2d import math, sys, random

def to_pygame(position):
return int(position.x), int(-position.y+screen_y)

def line_to_pygame(line):
body = line.body
point_1 = body.position + line.a.rotated(body.angle) point_2 = body.position +
line.b.rotated(body.angle) return to_pygame(point_1), to_pygame(point_2)

###options#### screen_x = 600
screen_y = 400
num_balls = 10
pygame.init()
screen = pygame.display.set_mode((screen_x, screen_y)) clock =
pygame.time.Clock()
running = True

space = pymunk.Space()
space.gravity = (0.0, -200.0)

#create the base segment
```

```
base = pymunk.Segment(pymunk.Body(),(0, 50), (screen_x, 0), 0) base.elasticity
= 0.90
space.add(base)

#create the spinner
spinner_points = [(0, 0), (100, -50), (-100, -50)]
spinner_body = pymunk.Body(100000, 100000)
spinner_body.position = 300, 200
spinner_shape = pymunk.Poly(spinner_body, spinner_points)
spinner_shape.elasticity = 0.5
spinner_joint_body = pymunk.Body()
spinner_joint_body.position = spinner_body.position joint =
pymunk.PinJoint(spinner_body, spinner_joint_body, (0, 0),

(0, 0))
space.add(joint, spinner_body, spinner_shape)

#create the balls
balls = []
for i in range(1, num_balls):

ball_x = int(screen_x/2)
radius = random.randint(7, 20)
inertia = pymunk.moment_for_circle(radius, 0, radius, (0, 0)) body =
pymunk.Body(radius, inertia)
body.position = ball_x, screen_y
shape = pymunk.Circle(body, radius, (0, 0))
shape.elasticity = 0.99
space.add(body, shape)
balls.append(shape)

while running:
for event in pygame.event.get(): if event.type == QUIT: running = False
screen.fill((0, 0, 0))
#draw the ball
for ball in balls:
pygame.draw.circle(screen,(100, 100, 100), to_pygame(ball. body.position),
int(ball.radius), 0)
```

```
#draw the spinner
points = spinner_shape.get_vertices()
points.append(points[0])
pygame_points = []
for point in points:

x,y = to_pygame(point)
pygame_points.append((x, y))
color = THECOLORS["red"]
pygame.draw.lines(screen, color, False, pygame_points)

#draw the line
pygame.draw.lines(screen, THECOLORS["lightgray"], False,
line_to_pygame(base))

space.step(1.0/50.0) pygame.display.flip() clock.tick(50)
```

As you can see, this code uses PyGame to handle the drawing of the graphics and PyMunk to work out how they move.
PyMunk works with spaces like the one set up in the following lines:

```
space = pymunk.Space()
space.gravity = (0.0, -200.0)
```

This creates a new object from the Space class and sets its gravity. We can then create objects and add them to the space. This example uses circles, segments (that is, lines), and a polygon that’s defined by a series of points. (Note, however, that on the Raspberry Pi, there’s a bug that means you can only add a single segment to a space.) There’s also a pin joint which, roughly speaking, makes the shape behave as though a single pin has been attached at that position, allowing it to pivot round, but not fall, due to gravity.

In the main loop, we call space.step(1.0/50.0), which tells PyMunk to move all the objects in the space by 1/50th of a second.
The one slightly confusing thing about using PyGame with PyMunk is that they use different coordinate systems. As you saw before, PyGame has the point 0,0 in the top-left corner, but Pymunk has it in the bottom-left. This means that to draw objects in the right place, you need to calculate the new y value. This is the purpose of the function to_pygame(position).

As well as defining the position of the various objects, in PyMunk you can also

define the various physical properties they have, such as elasticity and inertia. By tweaking these properties, you can define how your world interacts.

If you run the code, you'll see that PyMunk has done all the difficult tasks of working out the movement, such as handling collisions and calculating how the balls bounce off the floor and each other. The results are in Figure 5-6. However, it comes at a price—this takes much more processing power than our previous game. Whilst the Raspberry Pi can handle simple physics simulations, you need to use them sparingly; otherwise, they'll run too slowly. Using the raspi-config tool, you can overclock your Raspberry Pi, which will help simulations run faster.

Figure 5-6: The PyMunk physics engine takes the hard work out of simulating real-world interactions.

TIP

Raspberry Pis will, by default, run at 700MHz. This means they can execute instructions at a rate of 700,000,000 per second. For tasks that require a lot of computing power, you can overclock the CPU using the raspi-config tool. It can go up to 1,000MHz (or 1GHz if you prefer). However, not all Raspberry Pis will work well when overclocked, and they can become unstable. If your Pi starts freezing, try

reducing the overclocking.

We've only touched on the basics here, but hopefully it's enough to get you started. There are some samples in the PyMunk ZIP file that you downloaded earlier. Not all of them run well under Python 3, but they should give you more of a taste of what's going on. There is also some slightly outdated but otherwise good documentation at http://pymunk.googlecode.com /svn/tags/pymunk-2.0.0/docs/api/index.html.

Summary

After reading this chapter, you should know the following:

- PyGame is a module to help you create games in Python.
- Classes that extend Pygame.sprite.Sprite can draw images on the screen.
- Sprites are drawn inside a class's self.rect rectangle.
- You can also use this rectangle to detect collisions between objects.
- Parallax scrolling can be used to create a sense of depth with 2D graphics.
- PyGame can also handle audio.

- For more realistic physics, you can use a physics engine like PyMunk, but it will slow down the execution.

Chapter 6
Creating Graphics with OpenGL

THERE'S NO DENYING that 3D graphics look cool. The sense of depth they create uses more than 2D graphics, and allows you as a programmer to create richer worlds. However, this comes at a cost. Firstly, they take much more computing power to render than 2D graphics, and secondly they are significantly more complex to program.

You often find graphics cards (sometimes known as a Graphics Processing Units, or GPUs) on normal PCs. These provide additional processing power that the computer can use to render complex 3D scenes. Basically, these add a lot of processors that can handle floating-point maths very quickly. A quick look at a Raspberry Pi will tell you that there's no space to add a GPU because it doesn't have the same layout as a PC. Instead of a motherboard with a processor,

memory, and expansion slots, everything is enclosed on a System on a Chip (SoC). This is the largest square chip in the middle of the Pi. If you look carefully at it from the side, you'll see that it's made up of two layers. The top layer is the RAM, and the bottom layer does the processing.

The bottom layer isn't just a CPU (Central Processing Unit), though. In fact, the CPU is only a small part of it. It also contains a GPU that's far more powerful than the CPU. When you run normal programs, the GPU sits idly by while the CPU does all the work. When working with 3D graphics, though, the CPU can't handle it by itself, so you have to off-load some of the work to the GPU. In this chapter, we'll look at how to use OpenGL (the GL stands for Graphics Library) to create 3D scenes using the GPU.

We may as well be honest with you from the start: this is the most complex chapter in the book. There's no way around that. Using OpenGL requires some maths, a new programming language, and a host of new concepts. We'll take it slowly though, and explain everything as we go along. However, if you're just looking for a way to easily draw cubes in a 3D world, you can skip ahead to Chapter 8.

Getting Modules

There are two modules you'll need: PyGame and RPiGL. If you haven't already installed PyGame, follow the instructions in Chapter 4. RPiGL is available from https://github. com/stephanh42/rpigl. Use the Download Zip button to get a zip, then unzip and install it with the following (in LXTerminal).

```
unzip rpigl-master.zip
cd rpigl-master
python3 setup.py build
sudo python3 setup.py install
```

If this works, you'll now be able to run the demo programs, so try running the following:

```
cd demos
python3 bumpedspere.py
```

You should see a bumpy sphere spinning round (yes, there does appear to be a typo in the filename). If you get any errors, you'll need to fix them before moving on.

Creating a Spinning Cube

A spinning cube is the standard graphic for all new 3D programmers to try. It's simple enough to be easily understood, yet covers all of the basics, and it doesn't require vast amounts of data to build the 3D model.

NOTE
Recall that the book's companion website is at www.wiley.com/go/pythonraspi. To avoid potential typos, you can download and copy and paste the text into your IDE or code editor.

The full code is on the website as chapter6-spinning-cube.py. We'll go through it in stages, as there's quite a lot going on.
First of all, you need to set up the data:

```
vertices = [(0.0,0.0,0.0), (0.5,0.0,0.0), (0.5,0.5,0.0), (0.0, 0.5,0.0), 0.0,0.0,-0.5),
(0.5,0.0,-0.5), (0.5,0.5,-0.5), (0.0, 0.5,-0.5)]

indices_face_1 = (0, 1, 2, 0, 3)
```

The list vertices holds a list of all the corners of the cube we'll draw. You're actually going to draw a bit more than just a spinning cube. There'll be four of the six faces of a spinning cube, the edges of a static cube, and some points. Doing this, you'll learn the different ways of drawing things on the screen.

The tuple indices_face_1 holds a list of the points that you'll use for drawing the particular item on the screen (the number is the index of the list vertices). Here there are the indices (the plural of index) for one of the faces of the cube, but there are more in the actual code. You'll use these later to draw multiple items from the same pool of vertices.

The next bit is to set up OpenGL (note that if you're following along with the downloaded file, we're not going through it in order).

```
self.vertex_shader = glesutils.VertexShader(vertex_glsl) self.fragment_shader =
glesutils.FragmentShader( fragment_glsl)

self.program1 = glesutils.Program(self.vertex_shader, self.fragment_shader)
self.program1.use()

glClearDepthf(1.0)
glDepthFunc(GL_LESS) glEnable(GL_DEPTH_TEST)
```

```
glClearColor(0.5, 0.5, 0.5, 1)
```

Earlier, we said that the GPU is an extra processing unit that you can use to perform some of the maths needed for 3D graphics. In order to use it, we need to create a program for it, and this has to have source code. These programs are built of two parts, the vertex shader and the fragment shader. The variables vertex_glsl and fragment_glsl contain the code for this program-within-a-program (we'll look at them in detail later). In order to use them, you have to convert them into shader objects, then combine these shader objects into a program that you can run. You can have more than one OpenGL program within your Python program, and you switch between them using the use() method. Here, there is just one program, so there's just one use() call at the start.

The final four lines set up OpenGL. Firstly, there are three lines that set it to clear at a depth of 1.0, then the final line blanks the screen out to a mid grey. The four values of OpenGL colours are Red, Blue, Green, and Alpha (transparency). Each takes values in the range of 0 to 1. The next task is to load the data into the GPU.

```
self.verteces_buffer = array_spec.create_buffer( vertex_attrib=vertices)
self.elements_face_1 = glesutils.ElementBuffer( indices_face_1)
```

The main program is running on the CPU and the 3D modeling is taking place on the GPU. These are very close together on the Raspberry Pi, but it still takes a bit of time to transfer data between the two. Because of this, it's best to load as much information as possible into the GPU before you start running it. This is what buffers are for. Here you use two types of buffers to hold the vertex and index information. Once the program is running, you only need to send the identity of the buffer you're using. In this particular example, there are only a few items in each buffer, so it might not make too much difference. However, if you're loading complex 3D models, these could each hold huge amounts of information, and the lag in transferring them each time could be significant.

Vectors and Matrices

In the 3D worlds you create, every point is defined by a set of coordinates (x, y, z). The x is the horizontal position, y is vertical, and z is depth. In mathematical terms, this set of numbers is called a vector. Every point on an object is known as a vertex, and every vertex has a vector that describes its position. In order to

move objects around this 3D world, you need to manipulate these vectors. For example, you may want to zoom in on an object, which would mean moving every vector by a scaling factor. Or if you want to spin an object, you would need to move every vertex's vector accordingly.

This is done by vector-matrix algebra. In short, for every transition you want to make, you create a matrix (a square grid of numbers). You then multiply the vector by this matrix and you get a new vector.

Don't be confused by the word multiply; it's not like normal multiplication. The exact maths of what's going on is a little complex, and since OpenGL handles it all for you, you don't need to worry about it. All you need to know at this stage is that to move an object around, you create a matrix and multiply the vertices's vectors by the appropriate matrix. If you want to go beyond what we do in this chapter, it will be useful to learn more about what's going on, and there are plenty of resources, both online and in print, that can help you.

The following bit of code creates two matrices that show off the basics of how to do this. transforms.compose() is used to combine many matrices into a single one.

```
self.outer_matrix = transforms.compose( transforms.rotation_degrees(20, "z"),
transforms.rotation_degrees(20, "y"), transforms.rotation_degrees(20, "x"),
transforms.scaling(1.2))

self.points_matrix = transforms.compose( transforms.stretching(0.1, 1, 1.5),
transforms.translation(-0.5, -0.5, -0.5))
```

transforms.rotation_degrees() , transforms.scaling(), and transforms. stretching() each return a matrix that will perform the specified action when multiplied by a vector.

Before going on and drawing anything on the screen, let's now take a step backwards and look at the code we loaded into the GPU in the vertex and fragment shaders:

```
array_spec = glesutils.ArraySpec("vertex_attrib:3f")

vertex_glsl = array_spec.glsl() + """
uniform mat4 transform_matrix;
void main(void) {
```

```
gl_Position = transform_matrix * vec4(vertex_attrib, 1.0); gl_PointSize = 2.0;
}
"""

fragment_glsl = """ uniform vec4 color; void main(void) {

gl_FragColor = color; }
"""
```

This creates two strings, vertex_glsl and fragment_glsl, that contain code. However, as you've probably noticed, it's not Python code. Programs for the GPU have to be written in a special language called GLSL (Graphics Library Shader Language). It's similar to C (which is a programming language that you can write for the main CPU). The main differences with GLSL are

- Every statement has to end with a semicolon.
- The indent level doesn't matter, but code blocks are enclosed between curly braces.
- Variables have a type associated with them and can only hold data of that type.

There are also differences in the keywords and functions. Since this is all new, we'll go through it line-by-line.

```
array_spec = glesutils.ArraySpec("vertex_attrib:3f")
```

This creates a new ArraySpec object (which you'll need elsewhere). The parameter tells it that you'll pass the attribute vertex_attrib, which will be a series of three-dimensional float (i.e., 3f) vectors.

```
vertex_glsl = array_spec.glsl() + """
```

This creates the variable name and assigns the string to it. array_spec.glsl() just returns the code to properly create the attributes in array_spec (vertex_attrib). The three quotation marks tell Python that you're starting a multi-line string.

```
uniform mat4 transform_matrix;
```

This creates a new uniform variable called transform_matrix that's a four by four matrix. Theuniform keyword means that it can be set from the Python code.

```
void main(void) {
```

This creates the main function, which will be run every time the program is run. The first void means it doesn't return anything, and the second means it doesn't

take any parameters. Note the curly brace, which means you're starting the code block.

```
gl_Position = transform_matrix * vec4(vertex_attrib, 1.0);
```

The indent here isn't necessary (as it would be in Python), but is included because it makes the code easier to read. This is where you multiply the vector that describes the position of the vertex by the matrix that describes the transform. Notice that these are both fourdimensional. For now, don't worry about the last value and set it as 1.0.

Every vertex shader must set gl_Position as it's the variable that draws the vertex on the screen. Once it's set, OpenGL takes care of the rest.

```
gl_PointSize = 2.0; }
"""
```

gl_PointSize simply sets the size of points you draw on the screen (we'll cover points, lines, and triangles in a bit). The curly brace then finishes the main function, and the three quote marks end the string.

The vertex shader is called once for each vertex, while the fragment shader is called once for every point on the model. Therefore, the fragment shader is run far more times than the vertex shader, and you'll usually find that they're far simpler because of this. In this case, it's just four lines:

```
fragment_glsl = """ uniform vec4 color; void main(void) {

gl_FragColor = color; }
"""
```

Just as the vertex shaders always set gl_Position, fragment shaders always set gl_FragColor. This is a 4D vector that is the colour for that position.
With all this now in place, the only thing left to do is to place the items into the 3D world:

```
#Draw outer lines
self.program1.uniform.transform_matrix.value = self.outer_matrix
self.program1.uniform.color.value = (1, 1, 1, 1)
self.verteces_buffer.draw(elements=self.elements_outer,
mode=GL_LINE_STRIP) #Draw points
self.program1.uniform.transform_matrix.value = self.points_matrix
```

```
self.program1.uniform.color.value = (0, 0, 0, 1)
self.verteces_buffer.draw(elements=self.elements_points, mode=GL_POINTS)

#Draw spinning cube
rotation_matrix =
transforms.compose(
transforms.rotation_degrees(self.angle, "z"),
transforms.rotation_degrees(self.angle, "y"),
transforms.rotation_degrees(self.angle, "x")))

self.program1.uniform.transform_matrix.value = rotation_matrix
self.program1.uniform.color.value = (1, 0, 0, 1)
self.verteces_buffer.draw(elements=self.elements_face_1,

mode=GL_TRIANGLE_STRIP)
```

There are three different sets of draw functions, but they all follow the same format. Firstly, they set thetransform_matrix variable in the vertex shader with the attribute self. program1.uniform.transform_matrix.value. Then they use a similar line to set the colour variable in the fragment shader. Finally, they draw the points into the world with a call to self.verteces_buffer.draw(). This takes two parameters. Firstly, it takes the elements, which is simply the element buffer that contains the right indices of points, and secondly it takes a mode, which tells OpenGL what these vertices mean. The three here are GL_POINTS, GL_LINE_STRIP, and GL_TRIANGLE_STRIP.

GL_POINTS should be pretty obvious. It simply draws a point for every vertex. GL_LINE_ STRIP draws a continuous line and each vertex is a point on the line. There is also a mode called GL_LINES, which draws separate lines for every pair of vertices.

GL_TRIANGLE_STRIP draws a continuous chain of triangles. In this case we’re using five vertices to define a square. Actually, that’s not quite true: we’re using four vertices, but one of them is used twice. With triangle strips, the first three points have to make a triangle, then the third, fourth, and fifth, then the fifth, sixth, and seventh, then the seventh, eighth, and ninth, and so on. In each case, the last two points of the previous triangle make the first two points of the current triangle. In this way you can map out any surface.

There is also GL_TRIANGLES, which draws a triangle every three points and

GL_TRIANGLE_FAN, where every point shares a single point and they're fanned out a bit like the petals on a flower.

Bringing It All Together

The full code is as follows (remember, it's on the website as chapter6-spinning-cube.py):

```
import pygame
from rpigl import glesutils, transforms
from rpigl.gles2 import *

vertices = [(0.0,0.0,0.0), (0.5,0.0,0.0), (0.5,0.5,0.0), (0.0, 0.5,0.0),
(0.0,0.0,-0.5), (0.5,0.0,-0.5), (0.5,0.5,-0.5), (0.0, 0.5,-0.5)]

indices_face_1 = (0, 1, 2, 0, 3) indices_face_2 = (4, 5, 6, 4, 7) indices_face_3 =
(1, 5, 6, 1, 2) indices_face_4 = (0, 4, 7, 0 ,3)
indices_outer = (0, 1, 2, 3, 0, 4, 5, 1, 5, 6, 2, 6, 7, 3, 7, 4) indices_points = (0, 1,
2, 3)

array_spec = glesutils.ArraySpec("vertex_attrib:3f")

vertex_glsl = array_spec.glsl() + """
uniform mat4 transform_matrix;
void main(void) {

gl_Position = transform_matrix * vec4(vertex_attrib, 1.0); gl_PointSize = 2.0;
}
"""

fragment_glsl = """ uniform vec4 color; void main(void) {

gl_FragColor = color; }
"""

class MyWindow(glesutils.GameWindow):
def init(self):
self.angle = 10
self.vertex_shader = glesutils.VertexShader(vertex_glsl) self.fragment_shader =
glesutils.FragmentShader(fragment_glsl)

self.program1 = glesutils.Program(self.vertex_shader, self.fragment_shader)
```

```
self.program1.use()

glClearDepthf(1.0)
glDepthFunc(GL_LESS) glEnable(GL_DEPTH_TEST)

glClearColor(0.5, 0.5, 0.5, 1)
self.program1.uniform.light_dir.value = ((0, 1, -1)) self.verteces_buffer =
array_spec.create_buffer(vertex_attrib=vertices)

self.elements_face_1 =
glesutils.ElementBuffer(indices_face_1)
self.elements_face_2 =
glesutils.ElementBuffer(indices_face_2)
self.elements_face_3 =
glesutils.ElementBuffer(indices_face_3)
self.elements_face_4 =
glesutils.ElementBuffer(indices_face_4)

self.elements_outer =
glesutils.ElementBuffer(indices_outer)
self.elements_points =
glesutils.ElementBuffer(indices_points)

self.outer_matrix =
transforms.compose(
transforms.rotation_degrees(20, "z"), transforms.rotation_degrees(20, "y"),
transforms.rotation_degrees(20, "x"), transforms.scaling(1.2))

self.points_matrix =
transforms.compose(
transforms.stretching(0.1, 1, 1.5), transforms.translation(-0.5, -0.5, -0.5))

def on_frame(self, time):
self.angle = self.angle + time*0.02 self.redraw()

def draw(self):
#Draw outer lines
self.program1.uniform.transform_matrix.value =

self.outer_matrix
```

```
self.program1.uniform.color.value = (1, 1, 1, 1)
self.verteces_buffer.draw(elements=self.elements_outer,

mode=GL_LINE_STRIP) #Draw points
self.program1.uniform.transform_matrix.value = self.points_matrix
self.program1.uniform.color.value = (0, 0, 0, 1)
self.verteces_buffer.draw(elements=self.elements_points, mode=GL_POINTS)
#Draw spinning cube
rotation_matrix = transforms.compose

(transforms.rotation_degrees(self.angle, "z"),
transforms.rotation_degrees(self.angle, "y"),
transforms.rotation_degrees(self.angle, "x"))

self.program1.uniform.transform_matrix.value = rotation_matrix
self.program1.uniform.color.value = (1, 0, 0, 1)
self.verteces_buffer.draw(elements=self.elements_face_1,
mode=GL_TRIANGLE_STRIP)
self.program1.uniform.color.value = (0, 1, 0, 1)
self.verteces_buffer.draw(elements=self.elements_face_2,
mode=GL_TRIANGLE_STRIP)
self.program1.uniform.color.value = (0, 0, 1, 1)
self.verteces_buffer.draw(elements=self.elements_face_3,
mode=GL_TRIANGLE_STRIP)
self.program1.uniform.color.value = (0, 1, 1, 1)
self.verteces_buffer.draw(elements=self.elements_face_4,
mode=GL_TRIANGLE_STRIP)

MyWindow(200, 200, pygame.RESIZABLE).run()
```

The result should be a 3D rendering like the one shown in Figure 6-1.

Figure 6-1: This demonstrates the basic technique from which most 3D graphics are built.

Let There Be Light

When you run the previous example, you should see four sides of a cube spinning round as well as a few lines and some dots. However, you may notice that there's something missing: light. All the faces are the same brightness regardless of which way they're facing. In the real world, this rarely happens. Instead, you usually have one or more sources of light that illuminate the objects differently, depending on which way they're facing.

Some versions of OpenGL can handle this automatically. However OpenGL ES, the version on the Raspberry Pi (and most mobile devices) doesn't, so you have to calculate the lighting yourself.

The following example will create a spinning cube that is lit from a point light source. You'll see the various parts of the cube get brighter and dimmer as they move. There are two things that you use to calculate the brightness of a particular point—the distance of the point on the cube from the source of light, and the angle of the face compared to the light.

Calculating the Distance of the Point from the Light Source Let's look at the first of these. Each face on the cube is displayed by hundreds of pixels on the screen. The value of each of these pixels is calculated by the fragment shader. In order to calculate the colour of the pixel, the fragment shader needs to know how bright to make the pixel. This is done with the following code:

```
fragment_glsl = """ uniform vec4 color;
varying float brightness;

void main(void) {
gl_FragColor = brightness*color;
}
"""
```

As you can see, this uses two variables, color and brightness. brightness is a float that can be used to alter the value that gl_FragColor gets set to. You'll notice that they're different data types, but that isn't a problem. When you multiply a vector by a floating-point, you simply multiply every part of the vector by the number. For example, if you had the colour (0.8,0.8, 0.8) (which would be a light grey), and you multiplied it by 0.5, the resulting colour would be (0.4,0.4, 0.4) (a mid-grey).

brightness, then, should vary between 0 for no light and 1 for fully lit.

The two variables are declared differently. color is created with the keyword uniform, and brightness is set with varying. uniform variables are set in the main Python code (as you saw in the previous example). varying variables, however, are set in the vertex shader as shown here:

```
vertex_glsl = array_spec.glsl() + """
...
varying float brightness;

void main(void) {
...6;
float distance = length(vec4(light_position, 1.0) - gl_Position); brightness =
1.0/(distance * distance);

}
"""
```

This is missing the code to calculate the angles and position (which we'll look at later), but has everything to calculate how the brightness dims due to distance. The length() function returns the length of a vector (which can be calculated using the Pythagorean theorem). In this case, it's used to calculate the distance between the vertex and the light source.

As an object moves away from a light source, it gets dimmer. However, this doesn't happen linearly. If you move an object twice as far away, it doesn't get half as bright, it gets a quarter as bright. The relationship between the distance from a light source and brightness follows an inverse square. This is calculated in the final line.

This code, though, only calculates the brightness for the vertices. For the program to run properly, it needs to know what the brightness is for every point on the face. Fortunately, OpenGL takes care of this for you. Whenever you use the varying keyword to create a variable, it will interpolate the values passed to the fragment shader. This means that it will vary the value of brightness depending on the distance of the point it's rendering to the three vertices to create a smooth blend of brightness.

Calculating Reflecting Angles

The second aspect of brightness is the angle between the face and the source of light. To calculate this angle, you need a normal. A normal is a vector that sticks out of a face at 90 degrees.

Vectors can be used to describe positions, but they can also be used to describe lines. For example, the vector (1,1,1) could describe the point at those coordinates, or it could describe a line that's the same length and direction as the line from (0,0,0) to (1,1,1), but at any point in the 3D world. The vectors for vertices are position vectors, where as the vectors for normals are line vectors like this.

The normal is a vector to describe what direction the object is facing. Take for example the first face of the cube with vertices at (0.0,0.0,0.0), (0.5,0.0,0.0), (0.5,0.5,0.0), (0.0, 0.5,0.0). These vertices are all flat on the z-axis, so the normal is (0,0,1.0). If you drew a line from (0,0,0) to (0,0,1), it would be at 90 degrees to this face. Normals also always have a length of 1.

To calculate the brightness, then, you need to calculate the difference between this angle and the light from the light source. If they're exactly in line then the face should be fully lit, and if they're 90 degrees or more, the face shouldn't be lit at all. However, just as with the distance, there isn't a linear relationship between the two; instead the brightness varies in a sine curve. The sine function, however, would be the wrong way around, so you need to calculate the cosine.

For this, you need to reach into the trigonometric toolbox. There’s a function called the dot product that takes two vectors, and returns a number such that:
dot(A,B) = length(A) * length(B) * cosine(A,B)

Calculating the dot product is far quicker than calculating the cosine, so we can use this as a mathematical shortcut. It becomes simpler if both vectors have a length of 1. Then the dot product simply returns the cosine.

Normals always have a length of 1 so you can safely ignore that. There is also a function in GLSL called normalize() that takes any vector and returns one that’s in the same direction but that has a length of 1. You can calculate the cosine of the two vectors with:

cosine = dot(surface_normal, normalize(gl_Position - light_position)

However, most of the time there isn’t just one light source. There’s loads of light reflected off walls and other objects. Calculating all of this is phenomenally complicated, and the easiest thing to do is set an ambient light value. This is simply the amount of light that’s everywhere regardless of which direction the object’s facing. You can calculate the ambient with:

brightness = max(cosine, ambient_light)
All of this is then combined to make the vertex shader in the following code:

```
vertex_glsl = array_spec.glsl() + """ uniform mat4 transform_matrix; uniform
vec3 light_position;
uniform float ambient_light;
uniform vec3 face_normal;

varying float brightness; void main(void) {
gl_Position = transform_matrix * vec4(vertex_attrib, 1.0); vec4
spun_face_normal = normalize(transform_matrix *

vec4(face_normal, 1.0)); float distance = length(vec4(light_position, 1.0) -
gl_Position); vec4 light_direction = normalize(vec4(light_position, 1.0) -

gl_Position);
float light_amount_angle = max(dot(spun_face_normal, light_direction),
ambient_light);
```

```
float light_distance_drop = 1.0/(distance * distance); brightness =
light_amount_angle * light_distance_drop; }
"""
```

Note that this doesn’t run the dot product on the original surface normal, but on the surface normal that’s been transformed in the same way as the cube has. In this example, the transform_matrix won’t scale or stretch the normal so technically, it doesn’t need to be normalised. However, we’ve included this to make the code more useful in other programs.

The rest of the code is then basically the same as in the previous examples. There aren’t the points or lines, and there are all six faces of the cube. This is on the website as points or lines, and there are all six faces of the cube. This is on the website as lighting.py.

```
import pygame
from rpigl import glesutils, transforms from rpigl.gles2 import *

vertices = [(0.0,0.0,0.0), (0.5,0.0,0.0), x/4984720?c=pledges (0.5,0.5,0.0), (0.0,
0.5,0.0),
(0.0,0.0,-0.5), (0.5,0.0,-0.5),
(0.5,0.5,-0.5), (0.0, 0.5,-0.5)]

faces = [{"vertex_index":(0, 1, 2, 0, 3), "normal":(0,0,1), "colour":(1, 0, 0, 1)},

{"vertex_index":(4, 5, 6, 4, 7), "normal":(0,0,-1), "colour":(0, 1, 0, 1)},
{"vertex_index":(1, 5, 6, 1, 2), "normal":(1,0,0), "colour":(0, 0, 1, 1)},
{"vertex_index":(0, 4, 7, 0 ,3), "normal":(-1,0,0), "colour":(1, 0, 1, 1)},
{"vertex_index":(3, 2, 6, 3, 7), "normal":(0,1,0), "colour":(1, 1, 0, 1)},
{"vertex_index":(0, 1, 5, 0, 4), "normal":(0,-1,0), "colour":(0, 1, 1, 1)}]
array_spec = glesutils.ArraySpec("vertex_attrib:3f")

vertex_glsl = array_spec.glsl() + """ uniform mat4 transform_matrix; uniform
vec3 light_position;
uniform float ambient_light;
uniform vec3 face_normal;

varying float brightness;

void main(void) {
```

```
gl_Position = transform_matrix * vec4(vertex_attrib, 1.0); vec4
spun_face_normal = normalize(transform_matrix *

vec4(face_normal, 1.0)); float distance = length(vec4(light_position, 1.0) -
gl_Position); vec4 light_direction = normalize(vec4(light_position, 1.0)

gl_Position);
float light_amount_angle = max(dot(spun_face_normal,
light_direction), ambient_light);

float light_distance_drop = 1.0/(distance * distance); brightness =
light_amount_angle * light_distance_drop; gl_PointSize = 2.0;

}
"""
fragment_glsl = """ uniform vec4 color;
varying float brightness;

void main(void) {
gl_FragColor = brightness*color;
}
"""

class MyWindow(glesutils.GameWindow):
def init(self):
self.angle = 10
self.framerate = 20
self.vertex_shader = glesutils.VertexShader(vertex_glsl) self.fragment_shader =
glesutils.FragmentShader(fragment_glsl)

self.program1 = glesutils.Program(self.vertex_shader, self.fragment_shader)
self.program1.use()

glClearDepthf(1.0)
glDepthFunc(GL_LESS) glEnable(GL_DEPTH_TEST) glFrontFace(GL_CW)

glClearColor(0.5, 0.5, 0.5, 1)
self.program1.uniform.light_dir.value = ((0, 1, -1))

self.verteces_buffer =
```

```
array_spec.create_buffer(vertex_attrib=vertices)
for face in faces:
face["element_buffer"] =
glesutils.ElementBuffer(face["vertex_index"])

self.outer_matrix = transforms.compose( transforms.rotation_degrees(20, "z"),
transforms.rotation_degrees(20, "y"), transforms.rotation_degrees(20, "x"),
transforms.scaling(1.2))

self.points_matrix = transforms.compose(
transforms.stretching(0.1, 1, 1.5), transforms.translation(-0.5, -0.5, -0.5))

def on_frame(self, time):
self.angle = self.angle + time*0.02 self.redraw()

def draw(self):
self.program1.uniform.light_position.value = (0,0,-1)
self.program1.uniform.ambient_light.value = 0.3

rotation_matrix = transforms.compose(
transforms.rotation_degrees(self.angle, "z"),
transforms.rotation_degrees(self.angle, "y"),
transforms.rotation_degrees(self.angle, "x"))

self.program1.uniform.transform_matrix.value = rotation_matrix for face in
faces:
self.program1.uniform.color.value = face["colour"]
self.program1.uniform.face_normal.value = face["normal"]
self.verteces_buffer.draw(elements=face["element_buffer"],
mode=GL_TRIANGLE_STRIP)
MyWindow(200, 200, pygame.RESIZABLE).run()
```

The result is shown in Figure 6-2.

Figure 6-2: The light adds a sense of realism that isn't in the first program.

Making the Screen Dance

So far, you've seen how to draw cubes on the screen and light them, but not anything more than that. In the next project, you'll see how to create a 3D model that moves to music. Most computer music players have similar features that visualise the sound and provide a bit of video entertainment for the listener.

Since this is a chapter about 3D graphics and not audio processing, we'll make things simple for ourselves by only working with WAV files. This means we can use the wave Python module to extract the sound data from the file. If your music collection is stored as MP3 files, you'll need to convert one or more of them to WAV before continuing. You can do this with thempg123 command-line tool in LXTerminal. First you'll need to install it with:

sudo apt-get install mpg123
Then you can convert the files with:
mpg123 -w output-filename.wav input-filename.mp3
If you don't have any suitable music, you can download some legally from http://free musicarchive.org/.

The first step is to play the music. You do this in exactly the same way as you did the sounds in the previous chapter, using a PyGame mixer. This plays the sound through the Pi's audio channel, but it doesn't provide you with the sound

data needed to manipulate the 3D graphics. For this we'll use a second module, wave.

The following code can be used to import music from the file test.wav.

```
print("opening file")
sound_file = wave.open("test.wav",';rb')
print("getting parameters")
(channels, sample_size, frame_rate, frames, compression_type,
compression_name) = sound_file.getparams()

print("Number of channels: ", channels) print("Sample size: ", sample_size)
print("Frame rate: ", frame_rate)
print("Number of Frames: ", frames)
print("Compression type: ", compression_type) print("Compression name: ",
compression_name)

print("readframes")
data = sound_file.readframes(channels*sample_size*frames) print(len(data))
```

This creates the variable data, which contains the sound as a list of bytes. The sound, though, will be encoded in two-byte blocks. To read the value of a single block, you have to combine them using the from_byte() method of theint class.

```
sound_data = int.from_bytes(data[i:i+1],
byteorder='little', signed=True)
```

The data is stored in frames. Each frame is one of these two-byte blocks, and the value of each can vary between −32768 and 32767. For each point in time there are two frames if the sound file is in stereo. Typically, there are 44,100 of these frames every second (this is the number held in the frame_rate variable).
There are a number of ways that you could display this data. The only real requirement is that the output move in some entertaining way when the sound is playing. We're going to use two things. A set of 3D bars along the bottom of the screen that will plot out how the volume is changing, and a set of points (or stars if you're feeling poetic) that will flash to the music. Just to add some flair, we'll also make the whole visualisation spin around on the vertical axis, and blend the colour from blue at the bottom to red at the top.

Building the 3D Model

There are numerous ways we could model the data, but we're going to use the vertex data of a single cube and manipulate it with a transformation matrix for each bar on the chart. The stars will simply be a group of randomly positioned points. Let's deal with the blocks first.

Every block has the same set of vertices and indices. The difference between them is the size (which will simply be a stretching matrix), and the position (which will be a translation matrix). There will also be a couple more matrices that will be the same for every block: one to spin them around (which will change every frame), and one to zoom in and out (which will be set at the start).

You'll need to combine all these matrices. You could use the Python method transforms. compose() to do this. However, this needs to chunk through quite a bit of data every frame, and we mentioned at the start that the GPU is more powerful than the CPU. Therefore, it's more efficient to let the GPU handle all the matrix operations.

The shaders, then, are

```
vertex_glsl = array_spec.glsl() + """ uniform mat4 position_matrix;
uniform mat4 eye_matrix;
uniform mat4 scaling_matrix;
uniform mat4 sound_matrix;
uniform float point_size;

varying float red;
void main(void) {
gl_Position = eye_matrix * position_matrix * scaling_matrix * sound_matrix *
vec4(vertex_attrib, 1.0); red = (gl_Position[1]+0.9)/2.0;

gl_PointSize = point_size; }
"""
fragment_glsl = """ uniform vec4 color; varying float red;

void main(void) {
gl_FragColor = vec4(red, 0.0, (1.0-red)/5.0, 1.0);
}
"""
```

These are pretty similar to the ones you've seen before. It's worth noting that

matrix multiplication isn’t commutative. This is a fancy way of saying that the order in which you multiply the matrices does matter. It won’t always make a difference, but does sometimes. In this example, it’s important to make sure the eye_matrix (which should spin the entire scene) comes before theposition_matrix (which puts the individual bars in the right place). If they are the other way round then the individual bars will spin on the spot rather than the scene spinning as a whole. This isn’t necessarily a problem, we just felt it looked better like this. Getting the matrices in the right order is simply a case of thinking about which order you want the transformations to take place in (or, failing that, trial and error).

You’ll notice that there is a varying variable named red. This takes the vertical aspect of thegl_Position vector (which is y, or 1) and transforms it into a value for the colour.
You may also note that the size of the points is set by a uniform variable. This will be dealt with a little later.

Calculating the Sound Level

Every time a new frame is displayed, the program will calculate the current sound level and set one of the bars to that sound level. It’ll cycle through the bars in order so together they show how the sound level has changed over the past 20 screen draws.

In order to do this, you need to be able to calculate the sound level. This can be done using theon_frame() method.

```
scale_factor = 0
if frame_position + 1000 < len(data):
for i in range(1,500):
scale_factor1 = scale_factor +

int.from_bytes(data[frame_position+2*i: frame_position+(2*i)+1],
byteorder='little', signed=True)**2 scale_factor1 = scale_factor1 / 500

self.sound_matrix[self.counter%20] =
transforms.stretching(1.0,scale_factor1,1.0)
self.program1.uniform.point_size.value = float(scale_factor1/4)
self.counter = self.counter+1
```

For any position in the sound (which is calculated by seeing how much time has

passed since the track started playing), this checks the sound level for the next 500 frames and sums their squares. Squaring the sound level does two things. Firstly it makes sure all the values are positive since the square of a negative number is always positive. Secondly, it makes the blocks jump up far more with each increase in music, and this looks better onscreen. The point of this is to make something that looks pretty, not something that produces a scientifically correct graph, so manipulations like this are perfectly acceptable.

The reason it adds together 500 frames is simply because the more frames it adds together the more accurate it is, but also the slower it is. We found this to be a good compromise between performance (which corresponds to the frame rate of the display) and ability to accurately model the music. If you wish to make this perfectly correct, you'll need to introduce some timing functions like you saw in the previous chapter.

You'll notice that it also sets the size for the stars. The amount we shrunk the scaling factor by (500 for the bars and 4 for the stars) was determined by seeing what looked good, not by calculation.

The full code of the visualiser is as follows (it's on the website as chapter6-music.py).

```
import pygame
from rpigl import glesutils, transforms from rpigl.gles2 import *
import random
import wave
import time

vertices = [(0.0,0.0,0.0), (0.5,0.0,0.0), (0.5,0.5,0.0), (0.0, 0.5,0.0), (0.0,0.0,-0.5),
(0.5,0.0,-0.5), (0.5,0.5,-0.5), (0.0, 0.5,-0.5)]

indices_outer = (0, 1, 2, 3, 0, 4, 5, 1, 5, 6, 2, 6, 7, 3, 7, 4) vertices_points = []
indices_points = []

for i in range(0,100):
vertices_points.append(((20 * random.random())-10, (20 * random.random()),
(20 * random.random())-10)) indices_points.append(i)
array_spec = glesutils.ArraySpec("vertex_attrib:3f")

vertex_glsl = array_spec.glsl() + """ uniform mat4 position_matrix;
```

```
uniform mat4 eye_matrix;
uniform mat4 scaling_matrix;
uniform mat4 sound_matrix;
uniform float point_size;

varying float red;

void main(void) {
gl_Position = eye_matrix * position_matrix * scaling_matrix * sound_matrix *
vec4(vertex_attrib, 1.0);
red = (gl_Position[1]+0.9)/2.0;
gl_PointSize = point_size;

}
"""

fragment_glsl = """ uniform vec4 color; varying float red;

void main(void) {
gl_FragColor = vec4(red, 0.0, (1.0-red)/5.0, 1.0);
}
"""

class MyWindow(glesutils.GameWindow):

def init(self):
self.angle_x = 5 self.angle_y = 10 self.angle_z = 5 self.counter = 0
self.vertex_shader = glesutils.VertexShader(vertex_glsl) self.fragment_shader =
glesutils.FragmentShader(

fragment_glsl)

self.program1 = glesutils.Program(self.vertex_shader, self.fragment_shader)
self.program1.use()

glClearDepthf(1.0)
glDepthFunc(GL_LESS) glEnable(GL_DEPTH_TEST)

glClearColor(0.0, 0.0, 0.0, 1)
self.program1.uniform.light_dir.value = ((0, 1, -1))
```

```
self.verteces_buffer = array_spec.create_buffer( vertex_attrib=vertices)
self.points_buffer = array_spec.create_buffer( vertex_attrib=vertices_points)

self.elements_outer = glesutils.ElementBuffer( indices_outer)
self.elements_points = glesutils.ElementBuffer( indices_points)

self.blank_matrix = transforms.translation(0.0, 0.0, 0.0)
self.position_matrix = [] for i in range(0,20):

self.position_matrix.append(transforms.translation((i/10)-0.95, 0.0, 0.0))
self.sound_matrix = []
for i in range(0,20):

self.sound_matrix.append(transforms.translation( 0.0, 0.0, 0.0))
self.program1.uniform.scaling_matrix.value = transforms.scaling(0.1)
self.program1.uniform.eye_matrix.value =

transforms.compose(
transforms.rotation_degrees(self.angle_z, "z"),
transforms.rotation_degrees(self.angle_x, "y"),
transforms.rotation_degrees(self.angle_x, "x"), transforms.translation(0.0, -0.9,
0.0))

self.counter = 0

def on_frame(self, ftime): global start_time global data

self.angle_y = self.angle_y + .5

self.program1.uniform.eye_matrix.value = transforms.compose(
transforms.rotation_degrees(self.angle_z, "z"),
transforms.rotation_degrees(self.angle_y, "y"),
transforms.rotation_degrees(self.angle_x, "x"), transforms.translation(0.0, -0.9,
0.0))

frame_position = int((pygame.time.get_ticks() - start_time) * 44.1 * 4)
scale_factor = 0
if frame_position + 1000 < len(data):
for i in range(1,500):
scale_factor1 = scale_factor +
```

```
int.from_bytes(data[frame_position+2*i: frame_position+(2*i)+1],
byteorder='little', signed=True)**2

scale_factor1 = scale_factor1 / 500
self.sound_matrix[self.counter%20] =
transforms.stretching(1.0,scale_factor1,1.0)
self.program1.uniform.point_size.value = float(scale_factor1/4)
self.counter = self.counter+1

self.redraw() def draw(self):
self.program1.uniform.color.value = (1, 1, 1, 1) for i in range(0,20):

self.program1.uniform.position_matrix.value = self.position_matrix[i]
self.program1.uniform.sound_matrix.value =
self.sound_matrix[i]
self.verteces_buffer.draw(elements=self.elements_outer,
mode=GL_LINE_STRIP)
self.counter = self.counter +1

self.program1.uniform.position_matrix.value = self.blank_matrix
self.program1.uniform.sound_matrix.value = self.blank_matrix
self.points_buffer.draw(mode=GL_POINTS)

print("starting pygame mixer")
pygame.mixer.pre_init(44100, -16, 2, 2048) pygame.mixer.init()

music = pygame.mixer.Sound("tell.wav")
print("opening file")
sound_file = wave.open("test.wav",'rb')
print("getting parameters")
(channels, sample_size, frame_rate, frames, compression_type,
compression_name) = sound_file.getparams()

print("Number of channels: ", channels) print("Sample size: ", sample_size)
print("Frame rate: ", frame_rate)
print("Number of Frames: ", frames)
print("Compression type: ", compression_type) print("Compression name: ",
compression_name)

print("readframes")
```

```
data = sound_file.readframes(4*frames) print(len(data))

print("starting audio") music.play()
start_time = pygame.time.get_ticks() print("start time: ", time.clock())
MyWindow(200, 200, pygame.RESIZABLE).run()
```

The result, which has a surprisingly ’80s feel to it, is shown in Figure 6-3.

Figure 6-3: This looks much better when it’s moving. But don’t take our word for it. Run the code and see for yourself.

Taking Things Further

Consider these ideas for taking the project to the next level:

- Make the stars a constant size, but make them change in brightness. You could use a separate OpenGL program to do this, although it’s not essential.
- You could add lighting, although for it to make sense, you’d need to replace the lines with triangle strips.
- Experiment with the number of bars and stars. You should be able to make this easily configurable.

- Make the graph show the frequency breakdown rather than the volume over time. To do this, you’ll need to calculate the Fourier Transform of the music using the SciPi module. Note that this is quite a challenging and maths-y option

for interested readers.

■ Create a better interface for selecting which songs to play. This could be command-line, text-based, or graphical depending on your preferences. There are examples of all three elsewhere in the book.

■ Change the visualisation. Falling or swirling stars are one option. You could also investigate zooming in and out as well as rotating. The colour blend doesn't have to stay constant either. Be creative; the 3D world is there for you to mold into your imagination.

Adding Some Texture

We said at the start that this would be the most complex chapter in the book. Even though it has been, it's done little more than introduce OpenGL. There is much more that we haven't covered. If you've found it interesting and want to explore further, one of the most interesting things to look into are textures. These allow you to add more detail to your 3D models. There are a couple of examples that come with RPiGL to get you started. You can find them in the demos subfolder of the Python module that you downloaded at the start.

The first is icosa.py, and it takes a map of the world and projects it onto a spinning sphere. It takes the image from world_cube_net.png and loads it using the method glesutils. Texture.from_surface(), and then it uses bind() to make it available to the shaders.

The second is bumpedspere.py, and this works a little differently. Remember how you used the normal to calculate the amount of light reflected? Well, you can use this to make a smooth surface appear to have a texture. If you vary the normal so that it's not exactly perpendicular to the surface, it will vary the amount of light on it as though the surface is slightly bumped. This is really an optical illusion as there's no way that such a surface could exist in real life, but the effect is a surface that appears to have small bumps on it without the need for a vast web of triangles.

TIP

There is loads of really useful information on OpenGL in print and on the Internet. However, you need to remember that not all versions of OpenGL work

the same. The version of the Raspberry Pi is OpenGL ES 2.0, so you need to make sure that whatever you're trying works with this version.

Summary

After reading this chapter, you should understand the following a bit better:

■ OpenGL provides a powerful way of creating and manipulating 3D graphics. ■ However, with that power comes complexity.

■ Coordinates are stored as vectors, and you can manipulate them by multiplying the vector by a transform matrix.
■ If there are many transforms, then you can chain these multiplications together. However, the order in which you do this is important.
■ OpenGL runs as a separate program that is comprised of two shaders—the vertex shader and the attribute shader.
■ These shaders aren't written in Python but in the GL Shader Language or GLSL, which is similar to C.
■ In GLSL, every line has to end with a semicolon, and code blocks are enclosed in curly braces.

■ In GLSL uniform variables are set by the program and are constant for the entire draw() method call. varying vectors are set in the vertex shader and are interpolated for points on the surface.

■ OpenGL ES doesn't include any lighting, so you have to calculate it yourself using both the distance from the light and the cosine of the normal and the light direction. ■ Textures can be used to add detail to your objects.

Chapter 7
Networked Python

THE WORLD TODAY is more connected than it's ever been, and almost everything that you do on computers has some form of online component. The Raspberry Pi is no different. As long as you have a Model B, or a wireless USB dongle, getting your Pi connected to the Internet is trivial. There's the Midori browser you can use to surf the web, and mail clients are available. These are good for consuming content—getting information off the web and using services that other people have created. The power of the Raspberry Pi, however, lies in

creating things. With a few lines of Python, you can grab information off the web or use your Raspberry Pi to serve up content and services to the world. Read on to find out how.

Understanding Hosts, Ports, and Sockets

To communicate with another computer, you need to know where to send data to. It might be that you're just sending information to another computer in the same room, or you might be sending it halfway round the world. Regardless, you need to specify an address. The standard way of locating computers is by Internet Protocol (IP) address. There are two types of IP address, version 4 and version 6. At the time of writing, version 4 (IPv4) is almost universally used, so you'll read only about it. IPv6 addresses work in the same basic way, so you shouldn't have any difficulty using these should they become mainstream any time soon.

Locating Computers with IP Addresses

IPv4 addresses contain four numbers separated by dots. To determine your Raspberry Pi's IP addresses (it has more than one), open a terminal (not a Python session, but an LXTerminal session). And run ifconfig. This should output something like the following:

```
eth0 Link encap:Ethernet HWaddr b8:27:eb:f3:d5:23 inet addr:192.168.0.13
Bcast:192.168.0.255

Mask:255.255.255.0
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1 RX
packets:87523 errors:0 dropped:0 overruns:0 frame:0 TX packets:59811 errors:0
dropped:0 overruns:0 carrier:0 collisions:0 txqueuelen:1000
RX bytes:97997131 (93.4 MiB) TX bytes:12573160 (11.9 MiB)

lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
UP LOOPBACK RUNNING MTU:16436 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0 TX packets:0 errors:0
dropped:0 overruns:0 carrier:0 collisions:0 txqueuelen:0
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

This shows that there are two network interfaces: eth0 and lo. eth0 is the wired network connection, andlo is the loopback connection that just loops back to the same machine. In the previous example, the Ethernet connection has an IP address of 192.168.0.13, while the loop back is 127.0.0.1 (this is always the same).

As well as by IP addresses, we can also locate computers by hostname (such as www.google. com, or localhost). These are a convenient shorthand for IP addresses. When you use one of these, you computer connects to a name server and asks what IP address that hostname corresponds to, then connects to the returned IP address. Localhost is a slightly unusual case as it always corresponds to 127.0.0.1 and is therefore the local machine.

If you think of the IP addresses (or hostnames) as similar to building addresses in the real world, then within each building, you still have to address it to the right person. In a computer network, this is done with ports. If you have a piece of software that serves information, it will listen on a particular port. Clients can then connect to a particular port, and get that information. Some ports are used for particular services. For example, web servers generally listen on port 80, whereas SSH connections go over port 22.

In Python, sockets are objects that connect to a particular port on a particular host, or listen for a connection coming into a particular port. Once they're connected, you can send and receive data through the socket. That sounds a bit complicated, but it's actually quite simple in practice. The easiest way to understand it is with an example.

Building a Chat Server

Unlike most of the programs in this book, this program has two parts that need to be run at the same time: a client and a server. The server just sits and waits for a connection to come in, while the client establishes a connection. Here's the code for the server:

```
import socket

comms_socket = socket.socket()
comms_socket.bind(('localhost', 50000)) comms_socket.listen(10)
connection, address = comms_socket.accept()
```

```
while True:
print(connection.recv(4096).decode("UTF-8")) send_data = input("Reply: ")
connection.send(bytes(send_data, "UTF-8"))
```

You can see that this gets most of its functionality from the socket module. With a server socket, you first need to bind it to a port on the host. (Any of the ports above 50,000 should be free for temporary use. Actually, you can use pretty much any one that’s not currently in use, but it’s best to avoid the ones below 100 unless you’re sure they’re not being used.) listen() then sets it to wait for a connection. When a connection comes in, it moves onto the following line:

```
connection, address = comms_socket.accept()
```

This sets up a new socket (stored in the variable connection), which is connected to the client. You can now send and receive data over this new connection using the send() and recv() methods. These take streams of bytes not strings, so you have to convert back and forward between the bytes and UTF-8 (universal character set Transformation Format; it’s 8-bit) encoding that you use to display the information.

Here’s the code for the client:

```
import socket
comms_socket = socket.socket()
comms_socket.connect(('localhost', 50000)) while True:

send_data = input("message: ")
comms_socket.send(bytes(send_data, "UTF-8"))
print(comms_socket.recv(4096).decode("UTF-8"))
```

You can see that this time, instead of binding the socket to a host and port, the code connects to them. This time, the code doesn’t create a new socket, but sends and receives on the original one.

Other than this, the two programs are very similar. To run a chat, you’ll need two Python sessions, and the easiest way to open two is with two LXTerminal windows. In the first window, typepython3 server.py (if you’ve called the server server.py), and in the second, typepython3 client.py.

You’ll be able to pass messages back and forth between the two programs. Of course, these aren’t very networked. In fact, they’re running on the same

machine. There are places where it’s useful to run networking between two programs on the same machine, but generally with networking, you want to send data between two computers.

This code has all the basics needed to communicate between machines, but it just needs a bit of a menu to help users connect to the place they want to go. It’d also be easier if there was a single program that could handle both the client and server sides. An improved version of the chat program with these properties is as follows (you can find it on the website as program with these properties is as follows (you can find it on the website as chat.py):

```
import socket

def server():
global port
host = "localhost"

comms_socket = socket.socket() comms_socket.bind((host, port))
print("Waiting for a chat at ", host, " on port ", port)
comms_socket.listen(10) send_data = ""

while True:
connection, address = comms_socket.accept() print("opening chat with ",
address) while send_data != "EXIT":
print(connection.recv(4096).decode("UTF-8")) send_data = input("Reply: ")
connection.send(bytes(send_data, "UTF-8"))

send_data = ""
connection.close()

def client():
global port
host = input("Enter the host you want to communicate" +

" with(leave blank for localhost) ") if host == "":
host = "localhost"
comms_socket = socket.socket()

print("Starting a chat with ", host, " on port ", port)
comms_socket.connect((host, port))
```

```
while True:

send_data = input("message: ")
comms_socket.send(bytes(send_data, "UTF-8"))
print(comms_socket.recv(4096).decode("UTF-8"))

port = int(input("Enter the port you want to communicate on" + " (0 for
default)"))
if port == 0:
port = 50000
while True:
print("Your options are:")
print("1 - wait for a chat")
print("2 - initiate a chat")
print("3 - exit")

option = int(input("option :"))

if option == 1:
server()
elif option == 2:
client()
elif option == 3:
break
else:
print("I don't recognise that option")
```

You should recognise the networking code in this, and the rest you should be fairly familiar with. In order to communicate between two computers, you need to agree on a port, set one to listen for a connection, then connect from the other.

This approach does have a few problems. Firstly, the two chatters have to alternate messages, and secondly you can only communicate with people on your local network. Actually, this second problem depends on how your local network is set up. Remember that we said IP addresses were a little like building addresses? In many ways they are, and local area networks (LANs) are like towns. In the same way you can have two different buildings with the address “1 The High Street,” as long as they are in different towns, you can have two different computers with the IP address 192.168.1.2 as long as they are on

different LANs. There are three IP address blocks reserved just for local networks:

■ 10.0.0.0 – 10.255.255.255 ■ 172.16.0.0 – 172.31.255.255 ■ 192.168.0.0 – 192.168.255.255

Any IPv4 address that falls within these is a local-only address (this means you can only communicate with other computers on your local network), whereas ones that fall outside of it are public (which means any computer on the Internet can send data to them).

It is sometimes possible to connect to a local IP address from outside on the Internet. Whether this is possible depends on your Internet provider and router (look for the portforwarding settings). However, there are too many different setups for us to be able to provide much guidance here.

Tweeting to the World

Instead of doing battle with your ISP and router, there are a number of other ways to get around these two problems. By far, the easiest is to use another service to handle the messages for you. Twitter is just such a service; it handles text-message passing between one computer and the world.

Each Twitter user can send tweets of up to 140 characters long. If any of these tweets mention another Twitter user (all usernames start with an @), then the tweet will show up in their connections page. If you don't already have a Twitter account, you'll need one for this section. Even if you do have one, it's probably a good idea to get a new one so that you can try things out without sending test messages to all your followers. It's free and quite simple to get an account. Just head to www.twitter.com and follow the instructions. You'll also get a tour of the site, so it should all make a bit more sense after setting up an account.
Twitter is normally used by people sending messages via the website, but that's not the only way of doing it. They also have an Application Programming Interface (API) that allows you to send and receive messages from programs rather than from the web interface.

There's a Twitter module that makes accessing the Twitter API simple. It's not included in Raspbian by default, so you'll need to download it from https://github.com/sixohsix/ twitter/tree/master (use the Download Zip button on

the bottom right). Once you have it, open an LXTerminal session, then unzip and install the module with:

```
unzip twitter-master.zip cd twitter-master
python3 setup.py build
sudo python3 setup.py install
```

You're now almost ready to go, but as well as needing a Twitter username, you also need to register your application with Twitter to get the appropriate credentials for your application.

Once you've logged in to Twitter, go to http://dev.twitter.com, and select My applications from the user menu in the top-right corner. In the new screen, press Create An Application. On this form, you'll have to enter details of your program.

- The name has to be unique on Twitter, so test-app won't work. Be a little creative, or just mash a few keys to come up with something that hasn't been done before.
- The description can be anything as long as it's over 10 characters.
- The website doesn't have to be a website at all, it just has to look like one. We entered http://none.example.com.
- The callback URL can be left blank.

Other than that, you just have to agree to the Rules of the Road and enter the captcha.

The applications page has a series of tabs. You need to change one of the default settings, so go to the Settings tab, and switch Application Type from Read Only to Read and Write. This will allow you to post statuses as well as read information. Once this is done, press Update This Twitter Application's Settings.

Now you need to create an access token, so switch back to the Details tab, and click Create My Access Token. Once this is done, it'll take Twitter a few moments to update itself, and you may have to refresh the page for the section Your Access Token to appear. Once it refreshes, you're all set up and ready to go. You just need to get four pieces of information from the details page on this website: Access Token, Access Token Secret, Consumer Key, and Consumer Secret. These are all random strings that you'll need to copy and paste into the following example (see chapter7-twitter.py on the website):

```
import twitter

def print_tweets(tweets):
for tweet in tweets:
print('text: ', tweet['text'])
print('from: ', tweet['user']['screen_name'])

twitter_user = twitter.Twitter(
auth=twitter.OAuth("ACCESS-TOKEN","ACCESS-TOKEN-SECRET",
"CONSUMER-KEY","CONSUMER-SECRET"))
status = twitter_user.statuses

home = status.home_timeline() print("home")
print_tweets(home)

mentions = status.mentions_timeline() print("mentions")
print_tweets(mentions)

search_string = input("Enter text to search for, " + "or press enter to skip: ")
if search_string != "":
search = twitter_user.search.tweets(q=search_string) print("search")
print_tweets(search['statuses'])

tweet = input("Enter tweet, or press enter to exit: ")
if tweet != "":
twitter_user.statuses.update(status=tweet)
```

This little Twitter client isn’t the most user-friendly one available. In fact, it’s hard to imagine that anyone would use it over the website. However, it does have many of the different ways of interacting with Twitter that you may want to include in any applications you create. It should be fairly easy to adapt this code to your needs.

Weather Forecasts with JSON

In the previous example, the Twitter module provided all the basic functionality, but there won’t always be modules you can use. Sometimes you’ll have to write your own code to interact with web services. Fortunately there is a standard format for sending data back and forth than makes it easy to incorporate web services into your projects. JavaScript Object Notation, more commonly called

JSON, is that standard. Originally, it was designed to work with JavaScript, which is a programming language mainly used on web pages, but it also works well with Python.

OpenWeatherMap.org is a website that provides free access to weather forecasts that you can include in your software. It also happens to use JSON. To get a feel for what a JSON document looks like, point your web browser to http://api.openweathermap.org/ data/2.5/forecast/daily?cnt=7&units=meteric&mode=json&q=London. This is will return a seven-day forecast for London. It's not particularly easy to read, but you should notice that it looks like a Python dictionary that contains (amongst other things) a list of more dictionaries. Python can pull that information from the Internet using the urllib.request module in the following code:

```
import urllib.request

url = http://api.openweathermap.org/data/2.5/forecast/" + "daily?
cnt=7&units=meteric&mode=json&q=London"
req = urllib.request.Request(url)
print(urllib.request.urlopen(req).read())
```

This will grab the information from OpenWeatherMap.org and print it on the screen. However, the data is in a string. You can't simply access various parts of it as though they are dictionaries and lists even though they look like them. You could build a function to read through the string and split it up, but fortunately you don't have to. The json module can load it and return a dictionary that contains the various parts of it. For example:

```
import urllib.request, json

url = http://api.openweathermap.org/data/2.5/forecast/ "daily?
cnt=7&units=meteric&mode=json&q=London"
req = urllib.request.Request(url)
forecast_string = urllib.request.urlopen(req).read()
forecast_dict = json.loads(forecast_string.decode("UTF-8"))

print(forecast_dict)
```

You can now get any information you want out of the forecast_dict data structure. In the following example, we've built a simple weather forecast

program that prints out a sevenday forecast for a given city (see chapter7-weather.py on the website):

```
import urllib.request,json
city = input("Enter City: ")
def getForecast(city) :
url = http://api.openweathermap.org/data/2.5/forecast/ +

"daily?cnt=7&units=meteric&mode=json&q=" url = url + city
req = urllib.request.Request(url)
response=urllib.request.urlopen(req)
return json.loads(response.read().decode("UTF-8"))

forecast = getForecast(city)
print("Forecast for ", city, forecast['city']['country'])

day_num=1
for day in forecast['list']:
print("Day : ", day_num)
print(day['weather'][0]['description'])
print("Cloud Cover : ", day['clouds'])
print("Temp Min : ", round(day['temp']['min']-273.15, 1), "degrees C")
print("Temp Max : ", round(day['temp']['max']-273.15, 1), "degrees C")
print("Humidity : ", day['humidity'], "%")
print("Wind Speed : ", day['speed'], "m/s")
print()
day_num = day_num+1
```

Note that the metric unit for temperature is Kelvin. To convert Kelvin to Celsius, simply subtract 273.15. This example uses only some of the data that the API returned. Take a look at the forecast data structure to see what else is in there that might be useful to print out.

Using this same basic method, you should be able to work with any web APIs that support JSON. There’s a list of popular services at www.programmableweb.com/apis/ directory/1?sort=mashups, where you should find a way to get almost any information your applications need off the web.

Testing Your Knowledge

So far, you've seen how to query an API to get information out of it. Now it's time to test whether you've fully understood what's been going on. Have a go at the exercise that follows, and then refer back to the previous examples for anything you're unsure of.

Exercise 1

You can get the current weather for a city (such as London) using the URL http://api. openweathermap.org/data/2.5/weather?q=London. Use this URL to create a program that tweets the current weather for a location. See the end of the chapter for an example solution to this exercise.

Getting On the Web

So far you've seen how to pass data back and forwards between two computers, and how to send and receive data to and from an online API. The obvious omission in all this is websites. These are, after all, the most popular way of viewing information online. In this section, you'll learn how to use your Raspberry Pi to host a web page.

There are two parts to the web: HTTP and HTML. The former is Hypertext Transfer Protocol (the method that web browsers and websites use to communicate), while the latter is Hypertext Markup Language (the language that web pages are written in). Hypertext is just a fancy name for any text with links embedded in it. There are modules that'll handle HTTP, but you will need to learn a little HTML for this to work.

Modern HTML is a complex language that can be used to create powerful applications with all sorts of animations and interactions. However, the basics of the language are quite simple. Every web page is a separate HTML file, and every HTML file has two parts, the head and the body. The head contains various pieces of information about the page, while the body contains what's displayed on the screen. HTML uses tags to describe the different parts of the document. Almost all tags come in pairs with an opening tag (such as <h1>, which denotes a main heading) and a closing tag (such as </h1>). Tags are always enclosed in triangular brackets, and closing brackets start with a forward slash. The following example uses most of the basic tags (see chapter7-htmleg1.html on the website):

```
<!DOCTYPE html>
<head>
An example HTML file</title> </head>
<body>
<h1>An h1 heading</h1>
<h2>An h2 heading</h2>
<p>
A paragraph of text with a <a href=http://www.raspberrypi.org";link

<to the Raspberry Pi websites</a>
</p>
<p>
Another paragraph
</p>
</body>
```

There is much more to HTML than this, but since this is a book about Python rather than HTML, we won't go into more detail. This is more or less what you need for this chapter. If you're interested in learning more, there are loads of great resources for taking things further.http://w3schools.org is a good place to start.

Once you've saved that file, you can just open it on your computer, and it should open in a browser. This is fine for checking your pages, but it's no good for sharing your creation with the world. To do that you need the aforementioned HTTP. In Python, it's really easy to whip up a quick HTTP server. For example:

```
import http.server, os
os.chdir("/home/pi")
httpd = http.server.HTTPServer(('127.0.0.1', 8000),
http.server.SimpleHTTPRequestHandler) httpd.serve_forever()
```

This will start a web server running on your Pi with the root in your home directory. It'll run on port 8000. If you were paying close attention earlier, you'll remember that 80 is the default port for web servers. We're using 8000 here just in case you're running something else with a web server, but you can change it to 80 if you prefer.

Once you have the code running, you can point your Pi's web browser to

http://localhost:
8000/. Localhost always points back to the machine it’s running on (it’s linked to the IP address
127.0.0.1), and : 8000 means use port 8000 rather than the default port 80. You should see a rather underwhelming list of files here.

HTTP works with directories and files in exactly the same way your computer does. It starts from a root directory. In the case of this example, that root directory is /home/pi. In your web browser, you can either specify a directory (which ends with a /) or a filename. If you enter a directory, the web server first checks to see if there is a file called index.html. If there is, it displays that instead of the directory contents. Change the name of your HTML file to index.html, then point your browser to http://localhost:8000 again. This time you should see the file displayed.

Making Your Website Dynamic

The http.server module is great for quickly whipping up a server to share information, but it’s not very good at serving up information that changes. It does have some ways of adding a more interactive experience, but there’s a better alternative. The tornado module is designed to serve up content that’s created on the fly rather than content that’s stored in files. This makes it far more suitable for projects that need this extra versatility.

Unlike the previous example, which worked with the operating system’s file structure, Tornado creates its own virtual structure, and instead of HTML files, you create classes that output the appropriate HTML. First you need to install it by opening the LXTerminal and entering the following:

sudo apt-get install python3-tornado
Take a look at the following simple example. It just creates a website with "Hello World!" on it.
import tornado.ioloop import tornado.web
class MainHandler(tornado.web.RequestHandler): def get(self):
self.write("<!DOCTYPE html><head><title>" + "Hello world</title></head>" + "<body>Hello World</body>")
if _ _name_ _ == "_ _main_ _":
application = tornado.web.Application([

```
(r"/", MainHandler),
],)
application.listen(8888)
tornado.ioloop.IOLoop.instance().start()
```

There are two parts to this example. In the first part, we define the class that creates the web page, while the second part (which starts with the line if _ _name__ == "_ _main_ _":) defines and starts the web server. The main part of this is:

```
application = tornado.web.Application([ (r"/", MainHandler),
],)
```

This creates a new Tornado web application and sets the options for it. The main options are a list of tuples that define which class handles which pages. In this example, there's only one page, the root, or / and it's handled by the class MainHandler. There are other options that can be added here, which will be used in future examples. The final two lines just tell it to listen on port 8888 (so you can run it at the same time as the http.server server on port 8000), and set it running. With this running, you can point your web browser to http://localhost:8888 to see it work.

All the handler classes have to extend tornado.web.RequestHandler, which provides all the basic functionality. All you need to add is a get method that calls self.write() (or as you'll see later, self.render()); the superclass handles everything else.

It might seem a little strange to define a class, but not create any objects from it. This is because Tornado uses the class and handles the object creation for you.

You can use Tornado like this to serve up static content, but it's not very good. For starters, all the HTML is inside the Python code, so it's a bit messy. The real advantage of Tornado is when you start creating dynamic pages that can change. The next example will generate a specific greeting for the users. You can use the same file as the previous example, but create a new class with the following code:

```
class HelloHandler(tornado.web.RequestHandler): def get(self, name):
self.write("<!DOCTYPE html><head>
<title>Hello world</title></head>" + "<body>Hello " + name + "</body>")
```

Then alter the bottom section of the code to bring it in (changes are shown in bold):

```
if _ _name_ _ == "_ _main_ _":
application = tornado.web.Application([ (r"/", MainHandler),
(r"/hello/(.*)", HelloHandler),

],)
application.listen(8888)
tornado.ioloop.IOLoop.instance().start()
```

With these changes made and saved, run the code again. Point your browser to http:// localhost:8888/hello/Ben orhttp://localhost:8888/hello/Alex, or use your own name.

Using Templates

Being able to modify the HTML code like this obviously lets you create far more powerful websites than you could before. However, this still means including the HTML in the Python, which isn't pleasant. The solution to this is to use templates. These are HTML documents with bits of Python in them that Tornado uses to build the final page. A template for the HelloHandler would be

```
<!DOCTYPE html>
<head>
<title>Hello</title> </head>
<body>
Hello {{ name }}
</body>
```

As you can see, the Python variable to print goes inside double curly braces. Save this as hello-template.html in your home directory (/home/pi), and then update the HelloHandler class to be

```
class HelloHandler(tornado.web.RequestHandler): def get(self, name_in):
self.name=name_in
self.render("/home/ben/hello-template.html",

name=self.name)
```

Then rerun the program. You should get the same response, but you should also be able to see that this code is far cleaner and easier to maintain.

Sending Data Back with Forms

You've probably noticed, though, that not many websites get you to enter information through web addresses. This method is often done for page IDs (as you saw with the weather API).

HTTP has a method for sending back to the server: POST requests. So far, Tornado has only been serving GET requests (hence the method name). POST allows you to send more complex information back using HTML forms. Create a new HTML file with the following code:

```
<!DOCTYPE html>
<head>
<title>User Information</title>
</head>
<body>
<h1>User Information</h1>
<form action="/hello/" method="post">
Enter your name: <input type="text" name="name"> <input type="submit"
value="Sign in">'
</form>
</body>
```

Save it in your home directory as user-info.html. Now change the HelloHandler class to the following (changes are shown in bold):

class HelloHandler(tornado.web.RequestHandler): **def get(self):**
self.render(**"/home/ben/user-info.html"**) **def post(self):**
self.render("hello-template.html", name = self.get_ argument("name"))

The final code block also needs to be updated to reflect the new changes:

```
if _ _name_ _ == "_ _main_ _":
application = tornado.web.Application([ (r"/", MainHandler),
(r"/hello/", HelloHandler),

],)
application.listen(8888)
tornado.ioloop.IOLoop.instance().start()
```

Now run this, and again point your browser to http://localhost:8888/hello/. Roughly the same thing happens as before, but now you enter your name in the textbox.

The reason this works is the first time you go to http://localhost:8888/hello/, your browser sends a GET request, so Tornado renders user-template.html. When you click Sign In, your browser sends a POST request because the form has the action /hello/ (if an address doesn't have a server name, the browser sends it to the same server) and the method POST. This time Tornado renders hello-template.html. self.get_argument()then grabs the data the user entered. name corresponds to the name you gave the text input in user-info.html.

As well as variables, you can also put certain pieces of Python code inside a template. This code is enclosed between {% and %}. The most common pieces are
{% set my_var = 0 %}
{% for x in y %} ... {% end %}

As you can see, this is a little different from regular Python. You need the word set before a variable assignment, and rather than using indenting (which doesn't work with HTML), code blocks are finished with {% end %}.

As an example, you could change hello-template.html to the following:

```
<!DOCTYPE html>
<head>
<title>Hello</title>
</head>
<body>
{% set char_number = 0 %}
{% for letter in name %}
{{ letter }} <br>
{% set char_number = char_number + 1 %} {% end %}
{{ name }} has {{char_number}} characters </body>
```

This not hugely useful code, but it does a bit more than the previous template. It displays each letter of the name on a different line (
 is the HTML tag for a line break), and it also counts the number of letters.

Exercise 2

Create a new web app into which the users can enter a city, and then get a web page displaying a seven-day weather forecast (with data taken from openweathermap.org). An example solution is at the end of the chapter.

Keeping Things Secure

As you develop more and more powerful web applications with Tornado, sooner or later security becomes important. Even if you're only running it on a local network, you may have things that you don't want unauthorised people messing with. The final example in this chapter is a web app that provides some information about the Raspberry Pi.

This web app uses cookies for the login system. Cookies are bits of information that the web app stores in the browser. They can be used for a wide variety of reasons, but here we'll use them to track which sessions are logged in. If the users successfully log in, we set a secure cookie with their usernames. When they log out, we clear it. When they try to view the web app, the program checks if they have the cookie set, and if they don't, it redirects them to the login page. All the files for this are on the web page as chapter7-tornado.zip.

```
import tornado.ioloop import tornado.web
users = {"ben": "mypassword"}
class SysStatusHandler(tornado.web.RequestHandler):
def get(self):
if not self.get_secure_cookie("user"):
self.redirect("/login")
return
if self.get_argument("type") == "processes":

com = [["pstree"],["top", "-bn1"]]
elif self.get_argument("type") == "system":
com = [["uname", "-a"],["uptime"]]
elif self.get_argument("type") == "syslog":
com = [["tail", "-n100", "/var/log/syslog"]]
elif self.get_argument("type") == "storage":
com = [["df", "-h"], ["free"]]
elif self.get_argument("type") == "network":
com = [["ifconfig"]]
else:
```

```
com = [["df", "-h"], ["free"], ["uname", "-a"], ["who"], ["uptime"], ["tail',
"/var/log/syslog"],["pstree"], ["top", "-bn1"]]
self.render("sysstatus-template.html", commands=com)

class LoginHandler(tornado.web.RequestHandler): def get(self):
self.render("login-template.html")

def post(self):
print(self.get_argument("name"))
print(self.get_argument("password"))
if self.get_argument("name") in users.keys() and users[self.

get_argument("name")] == self.get_argument("password"):
self.set_secure_cookie("user", self.get_argument("name"))
self.redirect("/sysstatus?type=system")

else:
self.render("login-fail.html")
class LogoutHandler(tornado.web.RequestHandler):

def get(self):
self.clear_cookie("user")
self.render("logout-template.html")

if _ _name_ _ == "_ _main_ _":
application = tornado.web.Application([ (r"/login", LoginHandler),
(r"/sysstatus", SysStatusHandler),
(r"/logout", LogoutHandler),
], cookie_secret="put your own random text here') application.listen(8888)
tornado.ioloop.IOLoop.instance().start()
```

As you can see, the handler that does most of the work (SysStatusHandler) takes a page argument called type, which can take the following values—processes, system, syslog, storage, network, or all. It then sends a list of lists that corresponds to various Linux system commands in the template.

In HTTP GET requests, arguments are sent after a question mark in the URL, so http:// localhost:8888/sysstatus?type=network sends the argument type with the value network to thesysstatus page.

The template that does the hard work is sysstatus-template.html:

```
<!DOCTYPE html>
<head><title>Raspberry Pi System Status Checker</title></head> <body>
<h1>Raspberry Pi System Status Checker</h1>
<a href="sysstatus?type=processes">Processes</a>
<a href="sysstatus?type=system">System</a>
<a href="sysstatus?type=syslog">System Log</a>
<a href="sysstatus?type=storage">Storage</a>
<a href="sysstatus?type=network">Network</a>
<a href="sysstatus?type=all">Display all</a>
<a href="logout">Logout</a>

{% import subprocess %}
{% for command in commands %}

<h2> Command: {{command[0]}} </h2>
<h2> Output: </h2>
<pre>
{% for line in subprocess.check_output(command).splitlines() %} {{line}}
{% end %} </pre>
{% end %}

</body>
```

subprocess is a Python module that lets you run commands on the underlying operating system. In the case of the Raspberry Pi, that’s Linux. It takes a list as its input with the first item in the list being the command to run. Any subsequent items are the arguments passed to it. So, the list ["df", '-h"] runs the command df -h, which outputs the current disks on the system and information about where they’re mounted and how much free space they have. You can try it out in LXTerminal. In fact, you can try out all of the commands used here in LXTerminal.

The<pre> tags are for pre-formatted text. That is, it makes HTML respect the tabs and newlines in the text rather than just garbling it all together.
The other templates are straightforward. Here’s login-fail.html:

```
<!DOCTYPE html>
<html>
```

```
<head>
<title>Login Fail</title>
</head>
<body>
Your login has failed. Please click <a href="login">here</a> to try again.
</body>
```

Here’s login-template.html:

```
<!DOCTYPE html>
<head>
<title>Raspberry Pi System Status Login Form</title> </head>
<body>
<p>
Please enter your login information</p>
<form action="/login" method="post">
Username: <input type="text" name="name">
<br>
Password: <input type="password" name="password"> <br>
<input type="submit" value="Sign in">
</form>
</body>
```

Finally, here’s logout-template.html:

```
<!DOCTYPE html>
<html>
<head>
<title>Login Fail</title>
</head>
<body>
You have logged out. Please click <a href="login">here</a> if you wish to log
in again.
</body>
```

The login method we’ve created here does provide some security, but it is far from completely secure. If you create a web app that handles sensitive information, or allows the users some control over the Raspberry Pi, and it’s on a network where you don’t trust all the users, you need to properly investigate web

security. It’s far too big a subject for us to cover here, but a good place to start is the Open Web Application Security Project (see www.owasp.org).

Summary

After reading this chapter, you should understand the following a bit better:
■ Data can be sent back and forwards between two computers using the host and port number.
■ Hosts can be defined by either IP addresses or hostnames.
■ To connect to another computer, you need to use sockets, which allow you to send data.
■ APIs are interfaces programmers can use to access web servers and their data.
■ The Twitter API has a module that lets you easily retrieve and manipulate information from twitter.com.
■ APIs use a universal data format called JSON to make it easier to parse and share data between programs.

■ Web pages are written in HTML and served using HTTP.
■ http.server allows you to create a really simple web server.
■ You can use Tornado to create more complex web applications.
■ Tornado templates make it easy to render complex, dynamic information in HTML. ■ The server can receive information from users using HTML forms and POST requests. ■ Cookies allow you to store information, such as usernames, on the client’s browser.

Solutions to Exercises

Exercise 1

Here is an example program that tweets the weather to a particular location.

```
import twitter, urllib.request, json
twitter_user = twitter.Twitter(
twitter_user = twitter.Twitter(

lHOzvEpNA31LTiUCHkLSiqqW5Pbe7BvJKvKT2H6",
"AwYIcpfRFUt6F4hMoGfqMGINDiUrW49R1mjVqY6Bts",
"xStMTHb0HQZaEFLi2bsWA",
"YgFhiCpvlqLLe5Is5dRAZWNzlT84KnyZCMKfXIwN8"))

city = input("Which city? ")
```

```
url = "http://api.openweathermap.org/data/2.5/weather?q<=" url = url+city
req = urllib.request.Request(url)
forecast_string = urllib.request.urlopen(req).read() forecast_dict =
json.loads(forecast_string.decode("UTF-8"))

tweet_text = city + " weather is ' +
forecast_dict['weather'][0]['description']
twitter_user.statuses.update(status=tweet_text)
```

Exercise 2

There are many ways of doing this, but we used three files. Firstly, a template called cityinfo.html contains the following:

```
<!DOCTYPE html>
<head>
<title>Weather</title>
</head>
<body>
<h1>Which City?</h1>
<form action="/weather/" method="post"> Enter a city: <input type="text"
name="city"> <input type="submit" value="Submit">
</form>
</body>
```

Another template called weather-template contains:

```
<!DOCTYPE html>
<head>
<title>Weather</title>
</head>
<body>
{% set day_num = 0 %}
{% for day in forecast %}
<h2>Day : {{str(day_num)}} </h2>
<h3>{{day['weather'][0]['description']}} </h3> Cloud Cover :
{{str(day['clouds'])}} <br>
Temp Min : {{str(round(day['temp']['min']-273.15, 1))}} degrees C<br>
Temp Max : {{ str(round(day['temp']['max']-273.15, 1))}} degrees C<br>
Humidity : {{str(day['humidity'])}} %<br>
```

```
Wind Speed : {{str(day['speed'])}}m/s<br>
{% set day_num = day_num + 1 %}
{% end %}
</body>
```

Finally, the Python file contains:

```
import tornado.ioloop import tornado.web import subprocess
import urllib.request,json

class WeatherHandler(tornado.web.RequestHandler): def get(self):
self.render("/home/ben/city-info.html")

def post(self):
url = "http://api.openweathermap.org/data/2.5/forecast/" daily?
cnt=7&units=meteric&mode=json&q=" + self.get_argument("city")

req = urllib.request.Request(url)
response = urllib.request.urlopen(req)
self.forecast = json.loads(response.read().decode("UTF-8"))
self.render("weather-template.html",

forec ast = self.forecast['list'])

if _ _name_ _ == "_ _main_ _":
application = tornado.web.Application([ (r"/weather/", WeatherHandler),

],)
application.listen(8888)
tornado.ioloop.IOLoop.instance().start()
```

Chapter 8
Minecraft

FOR THE UNINITIATED , Minecraft is a game of survival where your character can build things and acquire materials by mining the world around him. It’s set in a world of blocks, and each of these blocks can be destroyed to reveal hidden treasures. Figure 8-1 shows a typical Minecraft world.

That’s a pretty inadequate description, but it should give you an idea of what’s

going on. There’s a full version that you can play on most computers, and there’s a special version unique to the Raspberry Pi. The special thing about the Pi version is that you can control the game through Python. Not only can you move your player, but you can also manipulate the entire world around you.

Figure 8-1: The 3D blocks have become emblematic of this game.

Exploring Minecraft

That’s enough trying to explain it, though. Minecraft is one of those things that you really have to see to understand. First of all, you’ll need to download the software from http:// pi.minecraft.net.

You should end up with a file called minecraft-pi-0.1.1.tar.gz. You can extract this archive through the terminal with this command:
tar zxvf minecraft-pi-0.1.1.tar.gz
Then, you can start the game with this command:
mcpi/minecraft-pi

NOTE

When you’re playing Minecraft, it will capture the mouse and keyboard. If you need to switch back to a non-Minecraft window, just press Alt+Tab, and it will release control.

Controlling Your Minecraft World

The first thing to do is start a new game and get a feeling for moving about in the world (use the mouse to change the direction you're facing, the left mouse button to attach something, and the w, a, s, and d keys to move around). You can build things by changing the tool you're using (pressing one of the number keys will select from the items displayed at the bottom of the screen), and then pressing the right mouse button to use it. These are just the very basic Minecraft controls, but they should be enough to give you a feel for how to work in the world.

To control the world, you need to use the Python API. This is included in the archive you downloaded earlier, but that version doesn't support Python 3. Instead, you need to downloadchapter8-minecraft-api.tar.gz from the book's website.

NOTE
The longer code examples and other appropriate APIs and downloads in the book are from the book's companion website at www.wiley.com/go/pythonraspi.
Once you have the new API, you need to extract it using the terminal command:
tar zxvf chapter8-minecraft.tar.gz

You'll need to change the directory using the terminal command cd into the new folder. Then you should create the programs for this chapter in that folder so they can pick up the Python APIs:

cd mcpi
The programs you write won't run Minecraft themselves, but instead connect to an instance of Minecraft that's currently running on the Pi.

With that in mind, start a new Python interpreter. Remember that it has to be in the

With that in mind, start a new Python interpreter. Remember that it has to be in the minecraft directory, and the easiest way to do this is open an LXTerminal session, navigate to chapter8-minecraft (use the previous cd line), and then enter python3.

The following code will connect to the currently running instance of Minecraft:
```
>>> import minecraft
>>> mc = minecraft.Minecraft.create()
```
You now have a Minecraft object called mc that you can manipulate. This object

holds a player object that you can move around. For example, try the following:
>>> mc.player.setPos(10,10,10)

You should find that the player moves. Your player will move to these coordinates (10,10,10), but since worlds are randomly generated, this may be in the air, in which case your player will fall until he reaches the floor. Alternatively, he may end up underwater, or underground. Try changing the coordinates to find him a spot of land. The first and last coordinates are the player's position on the ground (x and z), while the middle one (y) is his vertical position.

Creating Minecraft Worlds in Python

Once you've found a good position for your player, you can alter the world around him using thesetBlock(x,y,z,type) method.x, y, and z are the world coordinates and they are in the same format as for the player. type is the typeID of the block. Every different type of block has a typeID, which you'll need to use whenever you manipulate the world.

So, for example, if your player is at (10, 15, 22), you can create a mushroom next to her with the following:
>>> mc.setBlock(11,15,22, 40)
See Table 8-1 for a list of useful block types.

Table 8-1 Some Useful Type IDs
ID Block Type ID 0 Air 1 2 Grass 3 4 Cobble Stone 5 6 Sapline 8 10 Lava (will flow) 18 22 Lapis Lazuli 40 41 Gold Block 246

Building Worlds So far, you've been working with whatever world Minecraft created for you. However, if you're going to create any sort of reusable program, you need to be able to know where everything is located at the start. To do this, you need to set the world to be how you want it. The following script sets a 200 by 200 square in the centre of the world to be smooth grass (type 2), with air above it (type 0).

```
import minecraft
mc = minecraft.Minecraft.create()
mc.setBlocks(-100,-1, -100, 100, 0, 100, 2)
mc.setBlocks(-100, 1, -100, 100, 100, 100, 0)
```

Block Type
Stone
Dirt
Wooden Plank Water (will flow) Leaves
Mushroom
Glowing Obsidian

This uses the method setBlocks(x1, y1, z1, x2, y2, z2, type), which sets every block between (x1,y1,z1) and (x2,y2,z2) to be of the appropriate type.

Drawing Pictures
Now you know how to set blocks, you can start doing something useful with them. Were this a full version of Minecraft, you could use this to build some form of ultimate fortress to shelter your character. However, the version of Minecraft on the Pi isn't the full version. Instead, you can use this version for more aesthetic reasons, such as drawing pictures with blocks. The following code does just that (it's on the website as chapter8-art.py):

```
import minecraft
import sys
mc = minecraft.Minecraft.create()

height = 10
start_x = 0
start_y = 0
canvas = "horizontal"

r = 246 g = 18 b = 22

picture = [["R", "R", "R"], ["G", "G", "G"], ["B", "B", "B"]]

if len(sys.argv) > 1:
with open(sys.argv[1]) as f: picture = f.readlines()

pic_x = start_x
for line in picture: pic_y = start_y
for block in line:

if canvas == "horizontal": x = pic_y
y = height
```

```
z = pic_x

else:
x = height
y = pic_y
z = pic_x

if block == "R":
mc.setBlock(x,y,z,r)
if block == "G":
mc.setBlock(x,y,z,g)
if block == "B":
mc.setBlock(x,y,z,b)

pic_y = pic_y + 1 pic_x = pic_x + 1
```

This works in a similar way to the level-loading code from Chapter 4. It uses sys.argv to check if the user has specified a filename at the command line. If she has, it uses the text from that file to make a picture; otherwise, it uses the default, which draws three lines. There are then two for loops that iterate through every line, and every character in that line. If the loop encounters an R, G, or B it draws a red, green, or blue block, respectively.

However, you don’t have complete artistic control over the world. You can only create one of the predetermined block types, and not all of them will hover in the air. This example uses block type 246 (glowing obsidian) for red, type 18 (leaves) for green, and type 22 (lapis lazuli) for blue. Of course, you are free to change these to others if you want to create a different look.

The code for the coordinates is a little convoluted because it has two modes—vertical and horizontal—where it draws the picture in different orientations. Because of this, it has to manage two sets of coordinates: 2D ones for the picture and 3D ones for the Minecraft world. The if block translates between the two depending on which orientation is selected.

Notice that the program ignores any character that isn’t an R, G, or B, so you can include them in your picture, but they will be represented as space. For example, the following text will create a smiley face (see Figure 8-2):

```
-------
```

-G---G-
-G---G-
---B---
-R---R-
--RRR-

Figure 8-2: Smiling faces are just the start. You can use this method to draw whatever you want in the Minecraft world.

Taking Things Further

There are loads of ways you could expand this world. The simplest way is to expand the pallet from three colours up to as many as you have characters and block types for. A more complicated way of expanding this world would be to create 3D drawings. This will require you to change the file structure somehow. We recommend doing it as a series of 2D images with a specific line between them. So, for example, a smiley face two blocks thick would be:

-G---G-
-G-B-G-
---B---
-R---R-
--RRR-*

```
-------
-------
-G---G-
---B---
-R---R-
--RRR-
```

You could also use a similar file structure to define an animation, with each separate block being a separate frame.

Making the Game Snake

A key use of any 3D graphical environment is to make games. Minecraft itself is obviously a game, and you can also use it as a games engine to build more games. In this section, you see how to make the classic game Snake.

If you're unfamiliar with the game, you control a snake that moves around and eats apples. Every time it eats an apple, it grows one square longer. If it hits either itself or the walls of the level, it dies. The aim of the game is to get as long a snake as possible. It was one of the most popular games on mobile phones before the iPhone came along.

NOTE Remember that most of the longer code examples in the book are available for download from the book's companion website at www.wiley.com/go/pythonraspi. To avoid potential typos, you can download and copy and paste the text into your IDE or code editor. The code for the following example is chapter8-snake.py.

Code for the game is as follows (you can find it on the website as chapter8-snake.py).

```
import minecraft import pygame import sys
import time
import random

from pygame.locals import *
mc = minecraft.Minecraft.create() pygame.init()

#blank world
print("Resetting world")
```

```
mc.setBlocks(-100,-1,-100, 100, 0, 100, 2) mc.setBlocks(-100,1,-100, 100, 100,
100, 0) mc.player.setPos(0,2,0)

# create control window
display = pygame.display.set_mode((200,200))

#draw maze
print("drawing maze") height = 10

start_x = 0 start_z = 0
difficulty = 0.5 apple_freq = 2

picture = [["R", "R", "R", "R", "R", "R", "R", "R", "R"], ["R", "-", "-", "-", "-", "-
", "-", "-", "R"], ["R", "-", "-", "-", "-", "-", "-", "-", "R"], ["R", "-", "-", "-", "-", "-
", "-", "-", "R"], ["R", "-", "-", "-", "-", "-", "-", "-", "R"], ["R", "-", "-", "-", "-", "-
", "-", "-", "R"], ["R", "-", "-", "-", "-", "-", "-", "-", "R"], ["R", "-", "-", "-", "-", "-
", "-", "-", "R"], ["R", "-", "-", "-", "-", "-", "-", "-", "R"], ["R", "-", "-", "-", "-", "-
", "-", "-", "R"], ["R", "-", "-", "-", "-", "-", "-", "-", "R"], ["R", "-", "-", "-", "-", "-
", "-", "-", "R"], ["R", "-", "-", "-", "-", "-", "-", "-", "R"], ["R", "-", "-", "-", "-", "-
", "-", "-", "R"], ["R", "-", "-", "-", "-", "-", "-", "-", "R"], ["R", "R", "R", "R",
"R", "R", "R", "R", "R"]]

if len(sys.argv) > 1:
with open(sys.argv[1]) as f: picture = f.readlines()

posn_z = start_z
for line in picture: posn_x = start_x
for block in line: x = height y = posn_x z = posn_z

if block == "R":
mc.setBlock(x,y,z,246)
posn_x = posn_x + 1 posn_z = posn_z + 1
snake = [(int((posn_z - start_z)/2),int((posn_x - start_x)/2))] movement = [-1,0]
finished = False
grow_in = []
apple_in = apple_freq
while not finished:
for event in pygame.event.get(): if event.type == QUIT: finished = True
key_state = pygame.key.get_pressed() if key_state[K_LEFT]:
movement = [-1,0]
```

```
if key_state[K_RIGHT]:
movement = [1,0]
if key_state[K_UP]:

movement = [0,1] if key_state[K_DOWN]:
movement = [0,-1]

next_position = (height, snake[0][1] + movement[1], snake[0][0] +
movement[0])
next_block = mc.getBlock(next_position[0],
next_position[1], next_position[2])

if next_block == 246 or next_block == 22: finished = True

if next_block == 18:
grow_in.append(len(snake)) apple_in = apple_freq

snake.insert(0, (next_position[2], next_position[1])) for block in snake:
mc.setBlock(height, block[1], block[0], 22)

if 0 not in grow_in:
remove_block = snake.pop()
mc.setBlock(height, remove_block[1], remove_block[0], 0)

grow_in2 = []
for number in grow_in:
if number > -1:
grow_in2.append(number - 1) grow_in = grow_in2

if apple_in == 0:
apple_y = random.randint(1,int(posn_x - start_x)-1) apple_x =
random.randint(1,int(posn_z - start_z)-1) while mc.getBlock(height, apple_y,
apple_x) != 0:

apple_x = random.randint(1,int(posn_z - start_z)-1) apple_y =
random.randint(1,int(posn_x - start_x)-1) mc.setBlock(height, apple_y, apple_x,
18)
apple_in = apple_in - 1
time.sleep(difficulty)
mc.postToChat("Game Over")
```

There are two Minecraft methods that you haven’t seen before: getBlock() and postToChat(). Both of these methods are fairly self-explanatory. The first returns the type of block at a particular location, and the second displays text on the player’s screen.

You should recognise the first part of the code because it’s almost the same as the code you saw previously to draw pictures. It’s slightly simplified because here you need only one colour block for the maze that the snake moves through (we picked red, but you can substitute this for whatever you like).

The second part of the code (which includes the while loop and the five lines above it) contains the mechanics of the game. The snake is stored as a list of tuples. Each tuple contains the 2D coordinates for that section of the snake. These are then transformed into 3D coordinates and drawn on the screen.

Moving the Snake

The movement of the snake is done in two parts. Firstly, a new block is added to the start. This new block is calculated by adding the movement variable to the coordinate of the front of the snake. Secondly, the last position of the snake is removed from the list using pop(), which both takes it out of the list and returns it. The tile at this coordinate is set to 0 (air).

The key control is the same as you saw in Chapter 4. It uses PyGame. The only slight complication is that for this to work, there has to be a PyGame window, which must have focus in order to capture the keypresses. You should see a blank 200 by 200 window open when you run the program, and you’ll need to click on it in order to be able to control the snake.

Growing the Snake

Perhaps the most complicated part of the code is the part that controls how the snake grows when it eats apples. This is because the snake doesn’t get longer as soon as it eats the apple; instead, it gets longer when its whole body has passed through the block the apple was in.

To control this process, you use a list called grow_in. Every time the snake eats an apple, the length of the snake is added to the list. Each time the main loop is executed, every number in this list is reduced by one and negative numbers are discarded. This means that when the whole snake has passed through the apple,

there will be a 0 in this list. The final portion of the snake is removed only when there isn't a 0 in the list. All this is done in the following section:

```
if 0 not in grow_in:
remove_block = snake.pop()
mc.setBlock(height, remove_block[1], remove_block[0], 0)

grow_in2 = []
for number in grow_in: if number > -1:
grow_in2.append(number - 1) grow_in = grow_in2
```

Adding the Apples

The only other part left to do is to add the apples. A new apple appears exactly apple_freq loops after the snake ate the previous one. They're created at random locations inside the level with this code:

```
if apple_in == 0:
apple_y = random.randint(1,int(posn_x - start_x)-1) apple_x =
random.randint(1,int(posn_z - start_z)-1) while mc.getBlock(height, apple_y,
apple_x) != 0:

apple_x = random.randint(1,int(posn_z - start_z)-1) apple_y =
random.randint(1,int(posn_x - start_x)-1) mc.setBlock(height, apple_y, apple_x,
18)
```

This generates random positions, and then checks that there isn't already something in that position (if there is, it generates a new random position). The variable apple_in simply counts the number of loops since the previous apple was eaten.

Take a look at Figure 8-3 to see how it looks, or just run it for yourself.

Figure 8-3: You can make the game harder or easier by altering the map for the level. Just make sure there aren't any gaps that the snake can escape out of; otherwise, the snake can rampage unchecked through the Minecraft world.

Taking Things Further

As with the game from Chapter 4, this one isn't really complete yet, and we'll leave that up to you. Some ways it could be finished include:

- Some more graphics. For example, add an explosion when the snake dies, or add a countdown to the start. You could make the snake more graphically appealing, such as by using alternating blocks along its length.

- A score. This could be the length of the snake.
- A method to change the difficulty (use the *time.sleep()* on the loop). This could have a link into the score. It could even get faster as the snake gets longer.
- Surprises. For example, once you get past a certain length, the snake could change colour.
- Bonuses. These could appear on the map for a short period of time, and if the snake gets to them, the player gets bonus points.
- More levels. The maps could be different sizes, or they could have extra walls inside them to make it harder.

You don’t have to limit yourself to just improvements to Snake. You could use these same methods to make entirely new games. For example, the same basic mechanics could be used for games like Tron. (This game is inspired by the film. Install it with sudo apt-get xtron to find out more).

You could even build it into a platformer like the one in Chapter 4, or … well, you could do anything. You’re limited only by your imagination.

Summary

After reading this chapter, you should understand the following a bit better:
- Minecraft is a survival game set in a 3D world of blocks.
- There’s a special version of Minecraft for the Raspberry Pi and it allows you to control the world from Python.
- You can move the player using player.setPos(x,y,z).
- setBlock() and setBlocks() can be used to manipulate the world.

Chapter 9
Multimedia

IN THE PAST eight chapters, you’ve seen loads of ways of processing data, and getting it out of the Pi. However, so far you haven’t seen many ways of getting data into the Pi. There’s been some key and button pressing, and a bit with the mouse, but that’s about it. In this chapter, we’re going to change that. There are two additional sensors you can give your Pi to allow it to interact with the world in a far more natural way—microphones and cameras. We’ll look at each in turn and see how you can use them in your Python programs.

Using PyAudio to Get Sound into Your Computer

There are all manner of things you can do once you have sound in the computer, but they all start with the same process of getting sound in. The module PyAudio is the best one for quickly and simply getting audio into the computer. Once you’ve captured the audio, you can do whatever you want with it, but if you want to use it in other programs, you’ll need some format that other software understands. WAV is a simple lossless format that’s great for simple audio data (although it can produce huge files if you’re recording a lot of data).

You’ll also need a USB microphone to get the sound. The jack port of the Pi is only for sound output.
As always, the first thing you’ll need to do is install the modules with the following terminal command:

```
sudo apt-get install libportaudio0 libportaudio2 libportaudiocpp0 \portaudio19-dev python3-setuptools python3-pip
sudo pip-3.2 install pyaudio
```

Now let’s save some sound! The code for this is as follows (you can find it on the website as chapter9-record.py):

```
import pyaudio import wave

def record_sound(seconds, chunk_size, sample_rate, filename, channels, format_type):
p = pyaudio.PyAudio()

stream = p.open(format=format_type, channels=channels,
rate=sample_rate,
input=True,
frames_per_buffer=chunk_size)

print("Speak now")
frames = []

for i in range(0, int(sample_rate / chunk_size * seconds)): data = stream.read(chunk_size)
frames.append(data)
if i%int(sample_rate/chunk_size) == 0:

print(seconds - round(i/(sample_rate/chunk_size)), "seconds remaining")
print("Finished")

stream.stop_stream() stream.close()
p.terminate()

wf = wave.open("filename", 'wb')
wf.setnchannels(channels)
wf.setsampwidth(p.get_sample_size(format_type)) wf.setframerate(sample_rate)
wf.writeframes(b".join(frames))
```

```
wf.close()

chunk_size = 1024
sample_rate = 44100
seconds = 15
filename = "output.wav" channels = 1
format_type = pyaudio.paInt16 record_sound(seconds, chunk_size, sample_rate,
filename, channels,

format_type)
```

As you can see, there are a number of properties that you can alter to influence the recording. Some of these control how the data is captured and stored (for example, chunk_size and sample_rate), while others are more high level (like the number of seconds that you record for, or the number of channels you use). For now, just stick with these defaults unless you have a good reason to change them. Some of them are limited by the hardware, and PyAudio will throw an exception if you try to change them.

Recording the Sound

The record_sound() function does all of the hard work (it could be inline code rather than a function and it would work fine, but we make it a function here so that it is easier to include in other programs). The first and second lines create a new PyAudio object and then use it to open a stream. You get the audio data from this stream.

The main work of recording the sound is then done by the for loop:

```
for i in range(0, int(sample_rate / chunk_size * seconds)): data =
stream.read(chunk_size)
frames.append(data)
```

All this does is bring in the data in correctly sized chunks and save it into a list. Each of the chunks is a bytes data type, but you don’t need to worry about that.

Now that you have the audio data as a string of bytes, you need to do something with it. The simplest thing is to store it as a WAV file using the wave module, and that’s what the second half of the function does.

If you run the program, you'll first notice a string of errors. Don't worry about this; it's just PyAudio's way of letting you know everything that's going on. (They're thrown as PyAudio detects the audio hardware on the device. As long as there aren't any Python exceptions, everything's going fine.)

Then you'll have a file called output.wav in the directory you ran it from. This is a 15-second recording that you can play back with any audio software.

Speaking to Your Pi

There are all manner of things you can do with the audio you've recorded, but one of the coolest that doesn't require any musical talent is voice control. To do this, you need some way of processing the sound to extract its contents, and this is a surprisingly hard task.

Unfortunately, there aren't any modules that can make this process easy. However, as you saw in Chapter 7, Python modules aren't the only way of getting extra functionality for your program. Web services are another great source of features that you can easily plug into your software.

Google has an excellent speech-to-text web service that we can utilise here. It does have a couple of limitations, though. Firstly, the speech sections are limited to a maximum of 15 seconds, and secondly the audio clips have to be uploaded in FLAC format with a sample rate of 16,000.

The first rule shouldn't cause too many problems for voice control systems (although it rules out dictation software). However, the second restriction is a little awkward since there isn't a standard FLAC module for Python. There are a couple of options, but none of them is particularly suitable for our purposes. Here, we can turn to yet another source of extra features (that you'll learn more about next chapter) from the Linux command line. This allows you to run regular Linux programs and control them from inside your Python code.

The code to convert the speech to text (once it's FLAC-encoded; see later) is the following:

```
def google_speech_recognition(filename): global url

audio = open(filename,'rb').read()
http_header={'Content-Type': 'audio/x-flac; rate=16000'} request =
urllib.request.Request(url, data=audio,
```

headers=http_header)

response = urllib.request.urlopen(request) out = response.read()
json_data = json.loads(out.decode("utf-8"))

return json_data

This should be fairly familiar to you if you've read Chapter 7, so we won't go over everything again. The only new things here are the extra parameters on the call to urllib.request. Request(). The data parameter, fairly obviously, sends the data, which in this case is the audio file, and the headers parameter sends HTTP headers. These headers tell the web server about who is requesting the data, and what, if any, data they are sending. In this case, we're telling the server that we're sending FLAC-encoded audio with a sample rate of 16,000. This doesn't return the actual text, but returns JSON-encoded data that includes the text and the confidence that the text is right (with different accents and bits of slang, this is never going to be completely perfect). You can add a print line like the following if you want more information about what's returned:

print(json_data)
Again, there's more information on using this format in Chapter 7.

Processing text is far easier than processing audio, so you should now be able to integrate the output in your software. This example isn't yet complete, though. We're going to expand it so that you can ask it questions and, with a little luck, it should be able to answer them for you.

Asking the Program Questions

Answering questions is quite a challenging feature to add to a program, and would be a truly daunting task to code. Fortunately it's another area where you can get a little help. Wolfram Alpha is a web service that attempts to do just that. You can try it out in a web browser at www.wolframalpha.com. As you may have guessed, there's also an API to access it from a program.

Again, the code should be mostly familiar if you've read Chapter 7:

def wolfram_alpha(speech):
url_section = urllib.parse.urlencode(dict(input=speech,
appid=wolfram_alpha_app_id,

```
))

url = http://api.wolframalpha.com/v2/query?' + url_section
response = urllib.request.urlopen(url)
tree = ElementTree.parse(response) root = tree.getroot()

for node in root.findall('.//pod'):
print(node.attrib['title'])
for text_node in node.findall('.//plaintext'):

if text_node.text:
print(text_node.text)
print("****")
```

The one part that you may not understand is the second half. Wolfram Alpha returns the results as an XML file, and this code uses an ElementTree from thexml module to parse this and extract the relevant information. We won't go into this in detail here as it's not relevant to the rest of the chapter, but should you need to look into it in the future, you can find more information in thexml module's documentation.

NOTE

Before running this code, you need to get a Wolfram Alpha app ID. For this, you need to create an account at https://developer.wolframalpha.com/portal/signup.html. Once you have it, copy it to the appropriate place in the code.

Putting It All Together

You can combine these three functions (recording the sound, converting it to text, and getting answers to questions) to create a speech recognition oracle (which we call Pyri due to its similarity to a popular mobile-based application). You use the following code (it's on the website as chapter9-pyri.py). The functions discussed previously aren't repeated here, but you'll need to include them for the program to work.

NOTE

Remember that most of the longer code examples in the book are available for

download from the book's companion website at www.wiley.com/go/pythonraspi. To avoid potential typos, you can download and copy and paste the text into your IDE or code editor. The code for the following example is chapter9-pyri.py.

```
import urllib.request
import json
import pyaudio
import wave
from xml.etree import ElementTree import subprocess
import time

def record_sound(seconds, chunk_size, sample_rate, filename, channels,
format_type):
#Code give above

def google_speech_recognition(filename): #Code given above
def wolfram_alpha(speech): #Code give above

chunk_size = 512
sample_rate = 44100
seconds = 5
filename = "output.wav" channels = 1
format_type = pyaudio.paInt16

url = "https://www.google.com/speech-api/v1/recognize?" +
"client=chromium&lang=en-US"
wolfram_alpha_app_id = "PUT-YOUR-APPID-HERE"
record_sound(seconds, chunk_size, sample_rate, filename, channels,
format_type)

subprocess.call(["sox", "output.wav", "-r16k", "-t", "wav", "output-16k.wav"])
subprocess.call(["flac", "-5", "-f",
"output-16k.wav"],stdout=subprocess.PIPE )

time.sleep(2)

try:
google_data = google_speech_recognition("output-16k.flac")
except urllib.error.HTTPError:
```

```
print("voice recognition failure")
else:
print(google_data)

if google_data['status'] == 0:
wolfram_alpha(google_data['hypotheses'][0]['utterance'])
else:
print("Voice recognition failed. Try again")
```

The final try block may be a little unfamiliar to you because it includes an else clause. This just runs after everything else in the block, but only if the original code doesn’t raise an exception. In this case, that means as long as the Google request doesn’t throw an HTTPError.

Before running the code, you’ll need to make sure both of these are installed with the following:

```
sudo apt-get install sox flac
```

Then there are two system calls that convert the file to the appropriate format:

```
subprocess.call(["sox", "output.wav", "-r16k", "-t", "wav", "output-16k.wav"])
subprocess.call(["flac", "-5", "-f",
"output-16k.wav"],stdout=subprocess.PIPE )
```

sox is also known as the "audio Swiss Army knife." It has loads of features for dealing with sound files. In this case, it’s used to convert the file from a sample rate of 44,100 to 16,000. This is done with the -r16k option. The second line uses theflac program to encode the WAV file as a FLAC.

subprocess.call() is a feature we’ll look at in more detail in the next chapter, but for now, we’ll just say that it lets you run Linux commands from within Python.

Taking Things Further

If you feel like taking this example further, you could add it as a feature to the web browser you created in Chapter 4. Instead of displaying the text like it does here, it could take the browser directly to the relevant page.

You could also create a simple voice-driven menu so that you can use your program without a mouse or keyboard. For example, it could display a list of numbered options and the users simply need to say the number they want (a little like some telephone switchboards).

Making Movies

You should now have an idea of how to get started with audio on your Pi, so we'll move on to the second medium, video.

There are two types of cameras that you can attach to your Pi—USB webcams and the Raspberry Pi Foundation's camera module. We'll look at both here since they have different applications.

Using USB Webcams

Of the two, USB webcams are the best supported in Python. They also work the same on all computers, so we'll look at them first.

Oddly enough, the easiest way to capture images is using the PyGame module that you first looked at in Chapter 5. Previously, you used it to create a game, but the module mostly contains graphics features (which are often used in games), including ones to grab images from a webcam.

If you've already worked through Chapter 5, you'll have PyGame installed. If not, you'll need to do this now with the following:

```
sudo apt-get update
sudo apt-get install libsdl-dev libsdl-image1.2-dev \
libsdl-mixer1.2-dev libsdl-ttf2.0-dev libsmpeg-dev \
libportmidi-dev libavformat-dev libswscale-dev \
mercurial python3-dev
```

Then grab the latest version of Pygame with:

```
hg clone <span cssStyle="font-family:monospace"><span cssStyle="font-family:monospace">https://bitbucket.org/pygame/ pygame</span></span>

cd pygame
```

These two lines will download the current version of PyGame into a new directory called pygame, then move into it. Once it's there, you can build and install the module for Python 3 with:

```
python3 setup.py build
sudo python3 setup.py install
```

The simplest program is one that simply grabs a stream of images from the camera and displays them on the screen. Effectively, this turns your Raspberry Pi into a mirror. It's done as follows (the code is on the website as chapter9-mirror.py).

```
import pygame
import pygame.camera
pygame.init()
pygame.camera.init()

display = pygame.display.set_mode((640,480),0) cam =
pygame.camera.Camera("/dev/video0", (640,480)) cam.start()

while True:
image = cam.get_image() display.blit(image, (0,0)) pygame.display.flip()
```

As well as initialising PyGame, you also have to initialise the camera portion of the module. If you have just one camera connected, it should come up as /dev/video0; however, you may need to modify this if you have more than one camera.

Using the module then is as simple as calling the method get_image() on a camera object. This returns an image that can be used just like any other PyGame image. In this case, we'll just blit (draw) it to the main display, then call flip() to update the display (as covered in Chapter 5).

Once you have the image, you don't have to just spit it straight out to the screen. You can do other things with it as well. For example, there are some image effects you can apply, and you can send it to a file instead of to the screen.

Take a look at the following code (it's chapter9-mirror-lines.py on the website):

```
import pygame
import pygame.camera

pygame.init()
pygame.camera.init()
from pygame.locals import *

size = (320, 240)
display = pygame.display.set_mode(size,0)
```

```
cam = pygame.camera.Camera("/dev/video0", size) cam.start()

inverse = pygame.Surface(size, pygame.SRCALPHA) for i in range (1,50):
image = cam.get_image()
lines = pygame.transform.laplacian(image) inverse.fill((255,255,255,255))
inverse.blit(lines, (0,0), None, BLEND_RGB_SUB) display.blit(inverse, (0,0))
pygame.display.flip()

pygame.image.save(inverse, "/home/pi/test.png")
```

This has the same basic structure as the previous example, but with a few more operations. Firstly, it creates a new image called lines. This is made from a laplacian transform on the original image, which is a mathematical process that finds the edges in an image. The returned image will be white lines on black.

The program then inverts this image to create an image that's black lines on white. There isn't a method that just inverts the colours of an image, so to do this you have to create a white image and subtract the image from it (that is, call blit() with the special flag BLEND_RGB_SUB).

You may have noticed that this example doesn't have a while loop, but instead uses a for loop. This loop will execute 50 times and then exit. This should give you a few seconds of video (if you want to make it precise, you could update it with the timing functions used in Chapter 4). Once this is finished, it'll execute the final line, which saves the current inverted line image as a PNG file. This makes the program behave a little like a camera on a timer.

The final difference between this program and the first one is that it uses images that are a little smaller. The size is set in the size variable. This is because the transforms are quite computationally intensive, and reducing each axis of the image by half reduces the amount of processing by a quarter (because it is proportional to area). This makes the video far more responsive. You can change this to see how performance is affected.

Adding Computer Vision Features with OpenCV

PyGame is great for getting images into your computer, but isn't set up for doing much with them once it has them. There's another module called OpenCV that's designed for manipulating images. Unfortunately it didn't support Python 3 at the time of writing, although this should change in the future. Since it has some

pretty interesting features that you can't easily get without it, we're going to cover it using Python 2. Hopefully, an update to Python 3 will be available soon, and then you shouldn't have any difficulty using it in the newer version of Python.
First, install it with the terminal command:

```
sudo apt-get install python-opencv libopencv-core-dev
```

The first example is the same as the first example with PyGame: a simple webcam viewer. This should give you a feel for how the library works:

```
import cv2
capture = cv2.VideoCapture(0)

while True:
return_val, frame = capture.read() cv2.imshow('Face', frame)

key = cv2.waitKey(5) if key == 113:
break
```

This is structured roughly the same as with PyGame. You open the camera, then read a frame and display it. The final three lines listen for the users to press the q key (which corresponds to key 113), and exiting if they do.

OpenCV is a hugely powerful module for manipulating images. One of the most interesting features is object recognition. Your programs can identify certain parts of an image, for example an eye or a mouth.

This is all done with Haar cascades, which are saved as XML files. These are created from training sets of images that the computer compares to pick up similar objects. You can create your own using the opencv_traincascades program that runs on the Linux command line. However, you will need vast numbers of images (hundreds to get good results), both with the object you wish to recognise, and without it. There's a good tutorial on how to do this at http://docs.opencv.org/doc/user_guide/ug_traincascade.html.

We won't go into creating cascades, but OpenCV does come with some example ones installed and ready to go. The next example uses OpenCV bundles to recognise eyes and mouths. The code's on the website as chapter9-objectdetect.py.

```
import cv2
```

```
factor_down = 0.33 factor_up = 3
capture = cv2.VideoCapture(0) eye_data =
cv2.CascadeClassifier('/usr/share/opencv/haarcascades/'

+ 'haarcascade_eye.xml') smile_data = cv2.CascadeClassifier('/usr/share/opencv/
haarcascades/' +
'haarcascade_mcs_mouth.xml')

while True:

return_val, frame = capture.read()
gray_frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
small_gray_frame = cv2.resize(gray_frame, (0,0),

fx=factor_down, fy=factor_down)
eyes = eye_data.detectMultiScale(small_gray_frame, 1.3, 5)

for (eye_x, eye_y, eye_width, eye_height) in eyes: cv2.rectangle(frame,
(eye_x*factor_up, eye_y*factor_up), (eye_x*factor_up + eye_width*factor_up,
eye_y*factor_up+eye_height*factor_up), (255,0,0),2)

smiles = smile_data.detectMultiScale(small_gray_frame, 1.3, 5) for (smile_x,
smile_y, smile_width, smile_height) in smiles: cv2.rectangle(frame,
(smile_x*factor_up,

smile_y*factor_up),
(smile_x*factor_up + smile_width*factor_up,
smile_y*factor_up+smile_height*factor_up), (0,255,0),2)

cv2.imshow('Face', frame)
key = cv2.waitKey(5) if key == 113:
break
```

As you can see, loading the Haar cascades is simply a case of calling cv2.Cascade Classifier() with thecascade file as the sole parameter. This creates an object with a method called detectMultiScale(), which can be used to return a list of rectangles that match the cascade. There are a few more things going on here, though.

In the while loop, the program first captures a frame as in the first OpenCV

example, and then it creates two new images—gray_frame and small_gray_frame. These, unsurprisingly, are a greyscale version of the initial frame and a smaller version of the grey frame. You need to do this is because object detection takes quite a bit of computing power, and while the Raspberry Pi is capable of doing it on a full size image, it runs slowly if it does. The smaller the images is, the less accurate the object recognition is. We found that shrinking the original image to a third of its size was a good compromise. You can play around with the values of factor_down and factor_up (they must always be equal to the reciprocal of the other) to see what works best for you.

TIP

Remember that you can overclock your Pi using raspi-config in LXTerminal if you want better performance.

detectMultiScale() then returns the coordinates for all the rectangles that match the object, and the two for loops go through all of those returned and draws them on the image.

Taking Things Further

If you're prepared to take the time to create a set of training images, you can use this basic program in all sorts of programs. For example, you could create one that recognises the users rather than asking them to log in, or you could add OpenCV to a robotic buggy to make it understand simple signs.

Even without creating your own Haar cascades, you can add vision to your Pi. For example, you could use the hand Haar cascades and keep an eye on how the rectangles move. You could get it to watch for hand waving.

Using the Raspberry Pi Camera Module

USB webcams aren't your only option for adding images and video to your Raspberry Pi projects. In fact, they may not even be the most popular method for doing it. The Raspberry Pi Foundation released its own camera module, which plugs into the board. Here are some of the pros and cons of the camera module compared to a USB webcam.

Pros:

■ Better resolution for still images than most webcams (five megapixels) ■ Good command-line support for Linux

■ Competitive price (£19; about $31)
■ Infrared (night vision) version available
Cons:

■ Less support in non-Pi specific programs and Python modules ■ Limited to one camera per board
■ Short cable (15cm), although it is possible, but not easy, to extend it

The choice comes down to the specific project you need the camera for.

Camera Module and OpenCV

OpenCV, which you saw in the previous section, won't work out of the box with the Raspberry Pi Foundation's camera module. However, you can modify OpenCV to work with it by following the instructions at http://thinkrpi.wordpress.com/2013/05/22/ opencv-and-camera-board-csi/.

At the time of writing, there wasn't a Python module available for the Raspberry Pi camera board. One project that's working on it is available from https://github.com/ashtons/picam. It's still a little unstable. At the moment, the best approach is to use the command-line tool through Python to access the camera.

The first thing to do is make sure that the camera is connected and enabled. Open an LXTerminal session and enter:
raspistill -o test.png

This should display a preview of the camera output for a few seconds, then capture the result as test.png. If there are any problems, go back to the Raspberry Pi camera setup instructions at www.raspberrypi.org/camera and get this working before moving on.

This gives you the basic way of taking a picture with the command line. You can do the same in Python with:

```
>>> import subprocess
>>> subprocess.call(["raspistill", "-o", "test.png"])
```

If needed, you can then bring the image into Python by loading it as you would any other image. For example, with the image-loading features of PyGame that

you used in Chapter 4. raspistill allows you to set a wide range of options to control the way the image is captured. Enter raspistill at the command line for a full list of options. The options range from the simple, such as the size, to the more complex, such as metering mode or AWB.

As an example, the following code captures an image that's 200 by 200 pixels, has a high contrast, and uses the sketch image effect. In addition, it has no onscreen preview and a timeout of one millisecond.

```
raspistill -w 200 -h 200 -co 90 -n -t 1 -ifx sketch -o test.png
```

The result is a slightly cartoony image. This would be the perfect input to creating a timelapse cartoon video, so that's exactly what we'll do with the next example (it's on the website as chapter9-timelapse.py).

```
import subprocess import time
frames = 10

for i in range(1,frames):
filename = "test" + format(i, "03d") + ".jpg"
subprocess.call(["raspistill", "-n", "-t", '1", "-w", "200",

"-h', "200", '-co", "90", "-ifx", "sketch", "-e", "jpg", "-e", "jpg", "-o", filename])
print("taking photo", i)

print("Encoding. . .")
subprocess.call(["avconv", "-r", "5", "-i", "test%03d.jpg", "-r", "24", "-s",
"200x200",
"-vsync", "cfr" , "out.mpg"])
```

This code uses the -e jpg option onraspistill, which saves the image as a JPEG rather than as a PNG. This makes the conversion to a movie easier. Once it's taken all the images, it converts them to a movie using avconv. The key options here are the first -r. The number following this is the amount of images it will use to create one second of video. In this example, we've used five, but you can change this to whatever you like (the second -r is the frames per second in the final video). -i test%03d.jpg picks up files with names like test001. jpg and test002.jpg. This matches the line that creates the filename:

```
filename = "test" + format(i, "03d") + ".jpg"
```

These have to match each other so that the video encoder picks up the images that have been created. You can change test to anything you want, provided you change it in both places. If you want to take more than 999 images, you'll need to add more zeros. For example, if you wanted to take up to 99,999 images, you could change the threes in both lines to fives (the variable frames stores the number of frames captured (actually it will capture one less than this number, but we left it like this for simplicity).

-s 200x200 is the final dimension of the video in pixels. This doesn't have to match the size of the images. You can change out.mpg to whatever you wish to call the final video.
If you run this and create a video, you can then view the result with any standard video player such as VLC.

Creating Live Streams

The previous method works great if you want to create a video over a predetermined amount of time, but what if you want to keep a video going indefinitely? For example, what if you wanted to create a live video stream that you can watch over the Internet? Internet lore has it that the very first webcam was set up in Cambridge (the home of the Raspberry Pi) to allow people to keep an eye on a coffee pot, so they could time their visits to the canteen with a fresh brew. The following example re-creates that webcam.

In order to send data over the Internet, you'll need some form of server. There are a few that can do the job, but for the sake of familiarity, we'll use the Tornado server that you saw in Chapter 7.

The live stream can then be created with the following code. Before running it, you'll have to create the directory /home/pi/images with mkdir /home/pi/images.

```
import tornado.ioloop import tornado.web import subprocess import time

class MainHandler(tornado.web.RequestHandler):
def get(self):
subprocess.call(["raspistill", "-w", "200", "-h", "200",

"-e", "jpg", "-n", "-t", "1", "-o", "/home/pi/images/live.jpg"]) time.sleep(2)
self.write('<! DOCTYPE html><head>' +
'<META HTTP-EQUIV="refresh"' +
' CONTENT="5"></head><body>' +
```

```
'<img src="/images/live.jpg"></body>') class
ImageHandler(tornado.web.StaticFileHandler):
def set_extra_headers(self, path):
self.set_header('Cache-Control',
'no-store, no-cache, must-revalidate,' + ' max-age=0')

application = tornado.web.Application([
(r"/", MainHandler),
(r"/images/(.*)", ImageHandler, {"path":"/home/pi/images"})])

if _ _name_ _ == "_ _main_ _":
application.listen(8888)
tornado.ioloop.IOLoop.instance().start()
```

If this doesn’t look familiar to you, pop back to Chapter 7 and have a look at the section on Tornado. There are, however, a few new things in here.

In essence, all this does is wait for a user to request the page with the live stream (this is simply the root, but you can update it to be wherever you want), then it takes a new picture. The web page is then served, and it includes the picture. To make it a live stream rather than just a live image, the web page it serves includes the tag <META HTTP-EQUIV="refresh" CONTENT="5">, which tells the web browser to automatically refresh the page every five seconds. It’s not high-quality video, but it’s enough to keep you updated on the amount of coffee in a pot.

The class ImageHandler extendstornado.web.StaticFileHandler. It does this so that it can change the default settings on caching. By default, Tornado will try to be efficient and not reserve images it’s served before. That means if a web browser requests an image that Tornado has already sent to it, it’ll just tell it to use the same image. Since the browser is constantly checking the image live.jpg, it would normally keep reserving the first image. By adding this header, you’re telling it not to be so lazy and to actually reread the file each time.

This works, but it’s not ideal. For example, it takes a new picture every time someone requests a page, but if a lot of people are constantly requesting pages, it’ll get clogged up, as the camera can only take one picture at a time. Secondly, it doesn’t store any of the pictures. This isn’t a problem if you’re just watching coffee, but if you’re using this as a security camera for example, this could be an

issue.

To get around this, you need to separate the section that takes the images, and the one that serves them. The following code will constantly take images so they can by amalgamated into a video, and at the same time, it keeps live.jpg updated with the latest. It’s on the website as chapter9-live-take-pics.py:

```
import subprocess import time
file_number = 0
dir = "/home/pi/images/"

while True:
file_name = dir + format(file_number, "05d") + ".jpg" file_number =
file_number + 1
subprocess.call(["raspistill", "-w", "400", "-h", "400", "-e",

"jpg", "-n", "-t", "1", "-o", file_name]) subprocess.call(["cp", "-f", file_name, dir
+ "live.jpg"]) time.sleep(2)
```

The next piece of code starts the previous script automatically, and displays the latest image just as the previous example did. It’s on the website as chapter9-live-image-server.py.

```
import tornado.ioloop import tornado.web import subprocess

class MainHandler(tornado.web.RequestHandler): def get(self):
self.write('<! DOCTYPE html><head>' +

'<META HTTP-EQUIV="refresh"' + ' CONTENT="5"></head><body>' +
'<img src="/images/live.jpg"></body>')

class ImageHandler(tornado.web.StaticFileHandler):
def set_extra_headers(self, path):
self.set_header('Cache-Control',
'no-store, no-cache, must-revalidate, max-age=0')

application = tornado.web.Application([
(r"/", MainHandler),
(r"/images/(.*)", ImageHandler, {"path":"/home/pi/images"})])
```

```
if _ _name_ _ == "_ _main_ _":
subprocess.Popen(["python3", "chapter 9-live-take-pics.py"])
application.listen(8888)
tornado.ioloop.IOLoop.instance().start()
```

This uses subprocess.Popen() to create a new process that lets both of the programs run at the same time (more on this next chapter).

Taking Things Further

There are still a few problems though. This code will simply run constantly until it fills up the SD card with images. At this point, the Pi will probably crash. If you want to take things further, there are a few things you could do. For example, every certain amount of time (maybe every day or every 20,000 images), you could convert the images into a video, and then delete the images. For extra bonus points you could make the archive videos available online.

To make it more secure, you could constantly upload the videos to a central server that's offsite. This means that if your Pi is stolen or destroyed, you still have access to the surveillance footage.

If you are deploying this on a publicly accessible web server, you should implement some security so only authorised people are able to view the footage online. Take a look back to Chapter 7 for details on how to do this.

Summary

After reading this chapter, you should understand the following a bit better:

■ PyAudio is a great module for bringing sound into Python programs. ■ Thewave module lets you output WAVE files.

■ There aren't great Python tools for everything in audio, but that doesn't cause a problem because you can use the Linux command-line tools such as sox and flac to fill in the gaps.

■ You can also use web services such as Google's speech recognition to add even more audio features.
■ There are two options for adding cameras to your Pi: USB webcams and the Raspberry Pi Foundation's camera module.

■ USB webcams work best with standard Python tools such as PyGame and OpenCV. ■ Thecamera module works with OpenCV, but it also has its own set of command-line tools for grabbing images and videos.
■ OpenCV can be used to add computer vision features such as object recognition. ■ A web server such as Tornado can be used to serve streams of images on the Internet.

Chapter 10
Scripting

THERE ARE LOADS of different tasks that you need to perform on a computer to keep it running properly. For example, you need to back up your data regularly just in case there's a problem. Other tasks can depend on exactly what you use your computer for. You might need something to keep your music collection in order, or sort photos. This chapter looks at how you can use Python to make your life easier by automating these housekeeping jobs so that your computer keeps running with minimal intervention.

This requires a lot of interaction with the underlying operating system, so the first thing to do is learn a little about Linux.

Getting Started with the Linux Command Line

By this point, you're probably fairly familiar with general use of Raspbian. Raspbian brings together several distinct parts to create an operating system. Firstly, there's the Linux kernel. This is the bit that makes everything work at the lowest level. It's constantly running and manages the hardware and memory and controls how other programs run. Then there's the command line and a wide range of tools. These provide an entirely text-based way of using Raspbian, and it's an area that you'll be looking at in more detail in this chapter. The chances are, you probably don't use this too much at the moment; instead, you probably interact with the desktop environment, which is another distinct section of Raspbian. The default GUI is LXDE, and on top of this there are a wide range of graphical programs.

If you've mostly used Windows, the first big difference you'll notice will probably be the filesystem. You won't find a C drive (or for that matter, any other lettered drive) on your Raspberry Pi. Instead, everything is built into a

single hierarchy that starts at / (known as the root). Before getting started with Python scripting, you'll need to know where everything is in this filesystem, so open LXTerminal (the window that contains the text-based way of using Raspbian) and enter the following:

cd / ls

The first line changes the directory to / (the root), then the second lists everything in the current directory. You should get output like this (see Figure 10-1):

bin dev home lost+found mnt proc run selinux sys usr boot etc lib media opt root sbin srv tmp var

Figure 10-1: LXTerminal allows you to interact with the powerful Linux command-line environment.

These are the directories in the root (if there were any files, they'd be listed here as well, but there aren't any in this folder). If you use a different Linux system, you'll find a similar set of directories in /. Most of these directories you'll never need to touch. Raspbian will look after them for you, and keep everything up to date. The ones that you're likely to use are /home, where the users' home directories are kept, /media, where removable devices such as USB memory sticks will be found, and /etc, where system-wide application settings are kept.

It's important to realise that this filesystem doesn't directly correspond to a filesystem that's stored on a disk in the same way it does in Windows. So, for example, /home/pi is stored on the SD card. However, if you were to put in a USB memory stick called MyStick, you'll find it at /media/MyStick.

/sys and /proc don't exist on any disk. They're virtual filesystems that are created to look like normal directories and files, but are just the system's way for displaying information. For example, if you type:

cat /proc/cpuinfo
cat is a command that outputs the contents of one of more files, and /proc/cpuinfo contains the technical details of the CPU. You should get something like this:

```
pi@raspberrypi /proc $ cat /proc/cpuinfo
Processor: ARMv6-compatible processor rev 7 (v6l) BogoMIPS: 697.95
Features: swp half thumb fastmult vfp edsp java tls CPU implementer: 0x41
CPU architecture: 7
CPU variant: 0x0
CPU part: 0xb76
CPU revision: 7

Hardware: BCM2708
Revision: 000d
Serial: 0000000074f3d523
```

The environment in LXTerminal is known as Bash. It's a powerful environment and even has its own programming language. If you want to learn more about Raspbian and Linux in general, Bash is a good place to start, and there are loads

of resources (such as www.linuxcommand.org). However, this is a book about Python, so we'll leave it here and let interested readers learn on their own.

Using the Subprocess Module

The easiest way to interact with the underlying system from Python is to use the subprocess module. We used this in Chapter 7, so you may already be a little familiar with it. Open up a Python interpreter (if you're already using LXTerminal, you can do this just by typing python3), and enter the following:

```
>>> import subprocess >>> subprocess.call("ls")
```

As you can see, using subprocess.call(), you can run any command on the underlying OS. If there are spaces in the command you want to run, you need to separate the command into a list of strings. For example, the command cat /proc/cpuinfo becomes:

```
>>> subprocess.call(["cat","/proc/cpuinfo"])
```

This is all well and good, but all this really does is provide a more verbose way of running commands. After all, you could just have easily run the same commands in LXTerminal. As we said at the start, the aim of this chapter is to automate general tasks, and to do that you're going to need to read in the outputs so that you can manipulate them.

For example, cat /proc/cpuinfo returns a load of information, most of which you probably don't want to know. The following program strips out all the information except the line that tells you the type of processor the computer is running.

```
import subprocess
p = subprocess.Popen(["cat", "/proc/cpuinfo"], stdout=subprocess.PIPE)
text = p.stdout.read().decode()
for line in text.splitlines(): if line[:9] == "Processor": print(line)
```

This uses subprocess.Popen() rather than subprocess.call(). Doing this gives you much more control over what's going on because it creates a new object that you can use to get the information you want.
Whenever a command runs on a Linux machine, there are two pieces of output: stdout and stderr. Stdout (or standard out) is where all the normal output goes.

For example, when you run a Python program, any print() statements go to stdout. Stderr (or standard error) is where the system sends error messages. If you're running something in LXTerminal, both of these go to the screen, but when you're scripting things, it can be useful to split them up. That way, if you're running a lot of commands, you can send any error messages to one place so that you can see instantly if something's gone wrong without having to check through all the output.

The parameter stdout=subprocess.PIPE tells Python to keep stdout in the object we're creating rather than sending it to the screen. Since you're not telling it what to do with stderr, it will, by default, send that to the screen. So, if you change the line to:

```
p = subprocess.Popen(["cat", "/proc/cpuinfozzz"], stdout=subprocess.PIPE)
```

The error message will be printed on the screen even though it doesn't get printed in any print() statement.
If needed, you can also capture the stderr of a command. For example, take a look at the following program, which displays the contents of a file that the user enters:

```
import subprocess
f_name = input("Enter a filename: ")
p = subprocess.Popen(["cat", f_name ], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
text = p.stdout.read().decode() error = p.stderr.read().decode()
print(text)

if len(error) > 0:
print("*****ERROR*****") print(error)
```

Command-Line Flags

Linux commands often take flags. These flags come after the main command and are ways to tell it what to do. For example, in LXTerminal, try running the following:

```
ls
ls -a
ls --all
```

ls lists the contents of the current directory. By using the flag -a, you're telling it

to list everything in the directory (usually ls omits files and directories that start with a ‘.’). --all is the same as ls -a. Many commands have two versions of each flag—a short version that starts with - and a long version (often easier to remember) that starts with -. To get more information on how to use ls, run it with either the flag -h or-help.

Flags can also take values, as you’ll see in a minute.

If you’re developing scripts, you should try to follow these conventions whenever possible. There’s a module named optparse that can help. The previous example can be made to take its input from a flag rather than a prompted user input. Take a look at the following:

```
import subprocess
from optparse import OptionParser
parser = OptionParser()
parser.add_option("-f", "-file", dest="filename", help="The file to display")
options, arguments = parser.parse_args()

if options.filename:
p = subprocess.Popen(["cat", options.filename ], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)

text = p.stdout.read().decode() error = p.stderr.read().decode() else:
test = ""
error = "Filename not given" if len(error) > 0:
print("*****ERROR*****") print(error)
else:
print(text)
```

As you can see, this creates an OptionParser object. In this example, there’s only one option, but you can add as many as you like by having more calls to parser.add_option(). The first two parameters to this are the short and long versions of the flag. dest="filename" means that the value of the flag is stored in the attribute called filename of the options that are returned.

Note that the parser automatically creates the flags -h and --help, and builds the help text up from the help= parameters in add_options() calls.

If you save the previous code as print-file.py, you can run it in LXTerminal by

cding to the directory it’s saved in and entering python3 print-file.py -f /proc/cpuinfo. You can view the help with python3 print-file.py --help.

Regular Expressions

All this is, though, is a Python wrapper around cat that just does what the original does. It doesn’t actually add anything.
Let’s add a feature that lets the users specify which lines of the file they want to display.

Python (and many other programming languages) has a feature called regular expressions. This slightly oddly named feature (often shortened to regex) enables you to specify bits of text to match.

They do this with special characters. The most common special character is *. This means, match the preceding character zero or more times. For example: do*g will match dg, dog, doog, and so on. The character + will match the preceding character one or more times. For example,do+g will match dog, doog, dooog, and so on, but not dg. do?g, on the other hand will match dg ordog only.

A period will match any character other than a new line, so .* will match any line, while .+ will match any line that isn’t empty.
You can group characters together, so d[io]g will match dig and dog, and d[io]*g will matchdg, dog, dig, doog, dioioioig, and anything like that.
We’ll look at a few more features of regular expressions a bit later on, but for now let’s get started with using them. The following code is on the website as chapter10-regex.py.

```
import subprocess
from optparse import OptionParser import re

parser = OptionParser()
parser.add_option("-f", "-file", dest="filename", help="The file to display")

parser.add_option("-r", "-regex", dest="regex",
help="The regular expression to search for")
options, arguments = parser.parse_args()
if options.filename:

p = subprocess.Popen(["cat", options.filename ], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
```

```
text = p.stdout.read().decode() error = p.stderr.read().decode() else:
test = ""
error = "Filename not given"

if len(error) > 0:
print("*****ERROR*****")
print(error)

else:
for line in text.splitlines():
if not options.regex or (options.regex and re.search(options.regex, line)):
print(line)
```

This gets the regular expressions from the module re. There are two main ways of using regular expressions: re.match() and re.search(). The first one tries to match the regular expression from the start of the string, while the second one tries to find text anywhere in the string that matches the regular expression. This program does the latter because we wanted to make it as easy as possible to match lines.
There is a slightly convoluted condition in the if line:

```
if not options.regex or (options.regex and re.search(options.regex, line)):
```

This is to handle the case whereby the user hasn't entered a -r or-regex flag. It says to print the line if either there isn't a regex flag, or there is a flag and the line matches.
To match the first example, run it with:

```
python3 chapter.py -f /proc/cpuinfo -r Processor
```

Linux systems keep log files that register various events. These can be useful in diagnosing problems, but they can also get huge and be hard to work with. They're all located in the /var/log folder.syslog holds much of the general information about what's been going on. For example, if you're having some difficulty with a USB device, running the following log file list all the times Raspbian registered a new USB peripheral (either hub or device):

```
python3 chapter10-regex.py -f /var/log/syslog -r"USB.*found"
```

In this case, putting the regular expression in quote marks isn't necessary, but sometimes the Linux shell will try to process special characters before they're passed to Python. Using double quote marks stops that from happening, so it's a

good habit to get into when putting regular expressions on the command line.

Let’s return to features of regular expressions:

- [^abc] matches every character except a, b, and c, so d[^o]g matches dig, dxg, and dag but not dog.
- [a-c] matches the characters a through to c, so d[a-j]g matches dag, dbg, and dig, but not dkg ordog.
- {a, b} matches the preceding character anywhere between a and b times, so do{2,4}g matches doog, dooog, and doooog, but not dog ordoooooog.

In addition, there are a series of special letters that, when preceded by a slash, take on a special meaning. Table 10-1 lists these special letters.

Table 10-1 Escaped Characters in Regular Expressions

Character Description

\n The newline character
\t Tab
\d Any digit
\D Anything except a digit
\s Any whitespace such as a space, Tab, or newline \w Any alphanumeric character \W Any non-alphanumeric character

Obviously, this can lead to some problems if you want to match a \ character. For this, or any other time where you want to match a special character (such as ., * or+), you can escape them with a \. Therefore, \\w matches \w and \w matches any alphanumeric character.

This can create a problem when entering strings in Python, since slashes need to be escaped there as well. For example:

```
>>> print("\\") \
>>> print("\\\\") \\
```

NOTE

This isn’t a problem in this example because you’ve been passing the strings into Python, but if you’re creating strings for regex matching in Python itself, you’ll need to remember to double the amount of slashes you want to use in the regex.

Testing Your Knowledge

We’ve covered quite a log of regular expression information in quite a short space. To make sure everything is sinking in, take a look at the following exercise.
Take the following file:

aaa
a10
10
Hello Helllo Helloo

Which regex will match which lines? Try to work it out, then run the previous program on them to find out (you’ll need to create a text file in Leafpad or download it from the website, where it’s called chapter10-regex-test). For example, to check the first one, run:

python3 chapter10-regex.py -f chapter10-regex-test -r"."
Remember that the program uses re.search() notre.match()

■ . ■ \d ■ \D\d ■ l{3,4} ■ e* ■ e+ ■ [Ha] ■ \d{2,3} ■ 1?

Scripting with Networking

We looked at networking in Chapter 7, and we’re not going to repeat ourselves here. Instead, we’re going to look at ways you may need to use the network when scripting. The most common thing you’ll need to do is copy files between two computers. There is an excellent module for this called Fabric; however, at the time of writing, it doesn’t support Python 3. This is likely to change, but not in the immediate future.

In the absence of a module to handle the process, you could create the Python code from scratch and copy everything across. This is certainly possible, but it’ll be quite long winded. There is a Linux command-line program called scp (secure copy) that does this, and you’ve already seen how to run command-line programs.

There is a slight problem though. When you run scp normally, it’ll ask you for your password. This will cause a problem when scripting in Python because you can’t easily tell it to answer questions. scp does, though, allow you to set it up with certificates so that if you’re logged in as an authorised user on one machine,

you can log in without a password on another.
scp can be used to copy information between Linux computers (such as Raspberry Pis), so it will work between two Raspberry Pis, or between a Pi and a Linux server. Other (non-Linux) servers may also support it, but you'll have to ask your system administrator to set it up.

The first thing you need to do is log into the machine you want to transfer information to and run the following on the Linux command line, such as in LXTerminal. If you have only one Linux machine, you can try this out with a single computer, so run these commands on the same machine. If you only have remote access to this machine, you can do this via ssh.

```
ssh-keygen -t rsa
```

This will create two files in the .ssh folder in your home directory, called id_rsa and id_rsa.pub. These contain the public and private keys. id_rsa.pub is the public key, and you'll need to copy it across to the computer you wish to log in from. You can do this with a USB memory stick, cloud storage (such as Dropbox or Google Drive), or even email, but since we're talking about scp, you could do it with that. The format of an scp command is:

```
scp location1 location2
```

It simply copies the file from location1 tolocation2. If one of the locations is a normal file path, such as /home/pi/.ssh/id_rsa.pub, then scp deals with it on the local machine. However, if the location is in the form user@machine:/home/pi, then scp tries to log in as a user on the remote machine. machine can either be an IP address or a hostname. Typically with Pis, it'll be a IP address. So, if the IP address of the machine you want to copy your id_rsa.pub file is 192.168.0.10, and you want to use the user pi, the command would be

```
scp /home/pi/.ssh/id_rsa.pub pi@192.168.0.10:/home/pi
```

If you're trying this out on a single machine, you don't actually have to move the file, but just to try out scp, you can use the machine localhost. Therefore, the copy command is

```
scp /home/pi/.ssh/id_rsa.pub pi@localhost:/home/pi
```

Then you just need to copy the contents of the file into the authorized_keys file by logging into the machine you just copied the file to and running:

```
cat /home/pi/id_rsa.pub >> /home/pi/.ssh/ authorized_keys
```

That’s been a little fiddly, but you should now be able to copy files from this machine to the other one (but not the other way around) without using a password. To try it out, enter the following:

```
touch test
scp test user@machine:/home/pi/
```

Where user, machine, and /home/pi are changed as appropriate. The first line simply creates an empty file called test. If everything works correctly, you won’t need to enter a password.

With all that set up, you can copy files between machines using subprocess.call(). For example:

```
subprocess.call(["scp", "file1.py", "pi@localhost:/home/pi"])
```

Bringing It All Together

At the start of the chapter, we promised some Python that can make your regular computer chores easier, and while we’ve shown you lots of cool Python, we haven’t fulfilled that promise yet. Now we will. In this section, we’re going to create a Python program that can help you keep backup copies of your most useful files so that if disaster strikes, and your SD card breaks, you can get your data back.

We’ll do this using most of what we’ve covered so far in this chapter, and one more module as well, os. This provides access to some of the operating system’s functionality. Take a look at the following code (there’s an explanation after it) The code’s on the website as chapter10-backup.py:

```
import os
import tarfile
from optparse import OptionParser from time import localtime
import datetime
import subprocess
import re

parser = OptionParser()
```

```
parser.add_option("-f", "--file", dest="filename",
help="filename to write backup to (if no option is give, backup will be used)",
metavar="FILE")

parser.add_option("-p", "--path", dest="path",
help="path to backup (if no option is give, ~ will be used)")

parser.add_option("-v", "--verbose", action="store_true", dest="verbose",
default=False,
help="print status messages to stdout")

parser.add_option("-i", "--images", action="store_true", dest="images",
default=False,
help="backup image files")

parser.add_option("-c", "--code", action="store_true", dest="code",
default=False,
help="backup code files")

parser.add_option("-d", "--documents", action="store_true", dest="documents",
default=False,
help="backup document files")

parser.add_option("-a", "--all", action="store_true", dest="all", default=False,
help="backup all filetypes (this overrides c, d & i)")

parser.add_option("-m", "--mtime", dest="mtime", default=False, help="backup
files modified less than this many days ago")

parser.add_option("-r", "--regex", dest="regex",
help="only back up filenames that match this regex")

parser.add_option("-s", "--server", dest="server", default=False, help="copy
backup file to this remote point (should be an scp location)")

options, arguments = parser.parse_args()

if options.filename:
backup_file_name = options.filename + '.tar.gz'
else:
```

```
backup_file_name = "backup.tar.gz"

backup_tar = tarfile.open(backup_file_name, "w:gz")

file_types = {"code":[".py"],
"image":[".jpeg", ".jpg", ".png", ".gif"], "document":[".doc", "docx", ".odt",
".rtf"]} backup_types = [] all_types = False

if options.images:
backup_types.extend(file_types["image"])
if options.code:
backup_types.extend(file_types["code"])
if options.documents:
backup_types.extend(file_types["document"])

if len(backup_types) == 0 or options.all: all_types = True
if options.mtime:
try:
mtime_option = int(options.mtime)
except ValueError:
print("mtime option is not a valid integer.", "Ignoring option")
mtime_option = -1
else:
mtime_option = -1

if options.path:
if os.path.isdir(options.path):
directory = options.path

else:
print("Directory not found. Using ~") directory = os.getenv("HOME")

else:
directory = os.getenv("HOME")
for root, dirs, files in os.walk(directory):
for file_name in files:
if not options.regex or re.match(options.regex, file_name):

name, extension = os.path.splitext(os.path.join(root, file_name))
if (extension in backup_types) or all_types: modified_days =
```

```
(datetime.datetime.now() - datetime.datetime.fromtimestamp( os.path.getmtime(
os.path.join(root,
file_name)))).days
if mtime_option < 0 or modified_days < mtime_option: if options.verbose:
print("Adding ",
os.path.join(root,file_name), "last modified", modified_days, "days ago")
backup_tar.add(os.path.join(root,file_name)) if options.server:
subprocess.call(["scp", backup_file_name, options.server])
```

As you can tell from the numerous parser.add_option() calls, you can change this program to work the way you want it to. Its basic function is to copy files into a tar.gz file, which is a type of compressed archive that's popular on Linux systems. This file can then be copied automatically to a safe location on a separate server. Should anything then happen to the original files, you can resurrect them from this backup.

If you run python3 chapter10-backup.py -help, you'll get the following, which describes what it does:

```
Usage: chapter10-backup.py [options]

Options:
-h, --help show this help message and exit
-f FILE, --file=FILE filename to write backup to (if no option

is given, backup will be used)
-p PATH, --path=PATH path to backup (if no option is given, ~
will be used)
-v, --verbose print status messages to stdout
-i, --images backup image files
-c, --code backup code files
-d, --documents backup document files
-a, --all backup all filetypes (this overrides c, d &
i)
-m MTIME, --mtime=MTIME
backup files modified less than this many
days ago
-r REGEX, --regex=REGEX
only back up filenames that match this
regex
```

-s SERVER, --server=SERVER
copy backup file to this remote point
(should be an scp location)

The basic usage is
python3 chapter10-backup.py -f backup.tar.gz -p /home/pi -s \
pi@192.168.0.10:/home/pi/backups/

This tells the program to go through /home/pi and every subdirectory looking for files. It does this using the os.walk() function. This simply returns a collection of directories and files that you can move through using a for loop. This is done in the lines:

for root, dirs, files in os.walk(directory): for file_name in files:

The first line will go through every directory in turn and return the path to that directory (root), the subdirectories (dirs), and the files (files). Since this program only cares about files, there is only one inner loop that iterates through all the files. os.walk() automatically goes through all the subdirectories, so you don't need to direct it to do that.

There are then some options you can choose to limit which files get selected. The basic ones limit it by filetype. -i, -c, and -d limit it to just images, code, and documents, respectively, while -a overrides these and selects all files (the default). These don't select perfectly, but work based on the dictionary of filetypes:

file_types = {"code":[".py"],
"image":[".jpeg", ".jpg", ".png", ".gif"], "document":[".doc", "docx", ".odt", ".rtf"]}

If one or more of these is selected, it'll only back up files that end with extensions in the appropriate list. The program finds the file extension in the line:
name, extension = os.path.splitext(root, file_name)

The function os.path.splitext() (note that's split-ext notsplit-text) splits the filename into its two basic components, and returns these as two separate values. The first is the main part of the filename (which is captured in the name variable), and the part that comes after the final. (which is captured in the

extension variable). All that’s left to do is check whether the extension is in the list backup_types, which we made by joining the lists of specified types. If you use these options, you should make sure that the entries in the list cover all the file types you actually want to back up.
The-r or—regex flag can be used to specify regular expressions that filenames must match to be included in the backup. This is in the line:

```
if not options.regex or re.match(options.regex, file_name):
```

Note that this uses re.match() rather than re.search(). This means that the entire filename must match the regular expression. For example, the regex ".*\.py" will match all files with the extension .py (which is equivalent to using the -c flag).

The final option you can use is -m or--mtime, which is short for modified time. In other words, it backs up all files that were modified more recently than this number of days. It does this with the rather convoluted line:

```
modified_days = (datetime.datetime.now() -
datetime.datetime.fromtimestamp( os.path.getmtime(
os.path.join(root,file_name)))).days

if mtime_option < 0 or modified_days < mtime_option:
```

The part of this that does most of the work is os.path.getmtime(). This takes a filename complete with a path (that is, it needs /home/pi/filename rather than just filename), and it returns the timestamp when the file was last modified.

os.path.join() takes two arguments, a path and a filename, and it joins them to create what the previous function needs (you can’t just join two strings together because sometimes paths have a / on the end and sometimes they don’t).

The timestamp returned by os.path.getmtime() isn’t a regular date, but the number of seconds since January 1st 1970 (this was the standard for Unix systems, and is used on Linux systems as well). Therefore, to get the number of days between now and when the file was last changed, we first have to convert it into a Python datetime using datetime. datetime.fromtimestamp( ), and then take it off the current time.

With all that done, all that’s left is to see if the result is less than the number of days given as an option. The initial clause in the if (mtime_option < 0) is

because the program sets mtime_option to-1 if there’s a problem with what the user entered, or if there isn’t an mtime option.

These are all the options that limit whether a file is selected. Once they’ve all been checked, the only thing left to do is add it to the tar file. This is done using the module tarfile, which provides a really simple way to handle these archives. You just need to open the file at the start. This is done with the line:

```
backup_tar = tarfile.open(backup_file_name, "w:gz")
```

The second parameter (w:gz) specifies that the file should be opened for writing and that it’s a gzipped (that is, compressed) file. The appropriate files can then be added to this archive with:

```
backup_tar.add(os.path.join(root,file_name))
```

You should recognise the code that uses scp to copy the file to a remote server if the option is specified.

This program is one example of a script that can help keep your computer in order. It makes the task of creating backups trivial; however, you still have to remember to run it to create said backups. Python can’t help you here, but there’s a Linux feature called crontab that takes care of running programs at specific times. Because all the options of this program are on the command line, all you have to do is decide what you want to run and set chrontab to start it at the appropriate time.

There are two main options for crontab: -l displays the list of the programs it currently has set to run, and -e opens up a text editor where you can edit what programs run when. Each task to run is on a separate line, and consist of five numbers or asterisks separated by spaces followed by the command. The five numbers relate to the minute of the hour, hour of the day, day of the month, month of the year, and day of the week the command should run, and an asterisk means any. Take, for example, the following (as a single line):

```
0 0 1 * * python3 /home/pi/chapter10-backup.py -s
pi@192.168.0.10:/home/pi/backups
```

This will take a backup of your home directory (the default) at midnight on the first of every month. The following will run every day at midday:

```
12 0 * * * python3 /home/pi/chapter10-backup.py -s
pi@192.168.0.10:/home/pi/backups
```

Working with Files in Python

In this chapter, you've seen a lot of ways to deal with files. However, you haven't yet actually opened any of them in Python to read or write data to (except the tar archive, but that was a special case). In this section, you'll see how to store information in text files, and then read it back.

This is actually really easy. All you need to do is call the open() function. For example, to open the file myfile.txt and print out every line, you need the following:

```
file_a = open("myfile.txt", encoding="utf-8")
for line in file_a: print(line.strip())
file_a.close()
```

The open() function creates a file object. It can take a number of parameters. The essential one is the filename, and encoding is a particularly useful one since it tells Python what format the file is in. Most text files are utf-8, and that's the one you should use when creating your own files.

NOTE
Note that you can't use file as a variable name because it's used for other things in Python.

Once you've opened the file, you can loop through it with a for loop. The only slightly unusual thing here is the .strip() that we've called on line. This is because each line of the file contains a newline character which will be printed, and the print function then adds its own newline, so without this you'd get a blank line between each of the printed lines.

Once you've opened a file, you should always close it. There is another way you can write this code so that it automatically closes. That is

```
with open("myfile.txt", encoding="utf-8") as file_a: for line in file_a:
print(line.strip())
```

The two pieces of code do exactly the same thing. The with block will automatically close the file when the code block ends, so it's useful if you're prone to forgetting to close files. Writing to files is almost as easy. You just need to add a mode="w" to the parameters of open(), then you can write. Take a look at the following:

```
with open("myfile.txt", mode="w", encoding="utf-8") as file_a: for letter in "abcde':
file_a.write(letter + "\n")
with open("myfile.txt", encoding="utf-8") as file_a: for line in file_a:
print(line.strip())
```

This will overwrite myfile.txt. However, if you want to add to the end of the file, you can use mode="a" (append). This will leave the original text intact and add new information to the bottom of the file.

Summary

After reading this chapter, you should understand the following a bit better:

- The Raspberry Pi runs a version of Linux.
- Linux has a different filesystem to Windows, based around the root directory, /.
- Linux has an entirely text-based mode.
- You can run commands on this text-based mode using the subprocess module.
- Commands in Linux can output to stdout and stderr.

- When writing scripts in Python, it’s useful to get all the input as command-line flags if possible.

- Regular expressions are a way of matching patterns of text.
- scp can be used to copy files between computers.

- There are loads of functions in the os module to help you interact with the operating system (far more than we could cover here; take a look in the Python documentation for more information).

- Theopen() function can be used to open files for reading or writing.

Chapter 11
Interfacing with Hardware

UNLIKE MOST COMPUTERS, a Raspberry Pi has a series of General Purpose Inputs and Outputs (GPIOs) that allow you to interact with the world outside. These are the metal pins that stick up next to the SD card. You can use

them a bit like programmable switches to turn things on and off, or you can use them to get information from other sources. In short, they allow you to expand your Pi in any way you want. They're widely used by digital artists to create interactive displays and by robot builders to bring their creations to life. With a bit of imagination, there really is no limit to what you can achieve with the Pis GPIOs and a few components.

Since this chapter is all about controlling things outside of the Pi, you will need a bit more equipment to try the examples here. It needn't be expensive though, and you can get started for just a few pounds. Even as you improve, most of the bits you'll need are cheap and easily available both online and in hobbyist stores.

Setting Up Your Hardware Options

Before jumping in and building circuits, the first thing you'll need is a way of connecting to the GPIO pins on the Pi. Since you can't just connect wires straight to the pins (actually you can solder directly onto them, but it's not recommended), there are a few options for accessing them, covered in the next sections.

Female to Male Jumper Wires

These are probably the simplest option. They simply fit over the top of the GPIO pins and allow you to then connect them to a solderless breadboard. This is the simplest way to get access to the GPIOs. It's also the method you'll see in the pictures of this book. It's fine for connecting a few pins, but it can get a little confusing if you're accessing a lot of pins at once. Take a look at Figure 11-1.

Figure 11-1: These jumper wires have a female end that slots over the GPIO pins and a male end that you can fit into the solderless breadboard.

Pi Cobbler

The Pi Cobbler is a really simple design that takes the GPIO pins and connects them to a header that can be pushed into a solderless breadboard (see Figure 11-2). It doesn't add anything that you don't get by using jumper wires, but it's a bit tidier and it's less likely to get into a confusing knot of wires.

Figure 11-2: The Pi Cobbler gets all the GPIO pins onto the breadboard without risk of tangled wires or confusion.
© Adafruit Industries

Solderless Breadboard

You'll probably need one of these whichever option you go with. It's a way of connecting components to build circuits quickly, and allowing them to be taken apart again when you're finished. They come in different sizes, but they all follow the same basic layout. Down the long sides there are two parallel lines of pins that can be used as positive and negative rails. In between these, there are two banks of pins with a gap in the middle. These are connected in strips perpendicular to the long edge (see Figure 11-3). In the pictures, we'll be using a small solderless breadboard that doesn't have the positive and negative rails; however, you can do them on whatever size board you have. If you are interested in electronics, it's well worth getting a full-sized one as it will become the core of many of your projects.

Figure 11-3: With solderless breadboards, you can easily prototype circuits, and then dismantle them and build new circuits using the same components.
You can push most components straight into the holes, and make connections using either male-male jumpers, or pieces of single-core wire.

Stripboards and Prototyping Boards

Once you've prototyped your circuits on a solderless breadboard, you may wish to build them on a stripboard. This will permanently join all the components together, and is far more durable than a solderless breadboard. These are a little beyond the content in this chapter, but if you want to take things further, you'll probably soon find yourself using these.

An alternative is the prototyping board. This simply contains lots of holes you can solder into. There are no connections between the holes so you have to solder on whatever connections you want.

PCB Manufacturing

The most advanced option involves making your own PCBs (Printed Circuit Boards). There are a number of options for doing this, including commercial printing. This is for the final stage when you have a complete design. If you want to go down this route, Fritzing is an excellent resource (see http://fritzing.org). They produce software to help you design the boards, and a service to print them.

Getting the Best Tools

You won't necessarily need any tools to build simple circuits (as long as you have male-male jumpers for the breadboard), but there are a few that'll make life easier for you.

Wire Cutters/Strippers

These are pretty self-explanatory. You'll need these (they usually come as a single tool) if you're planning on using single-core wire to make connections. However, if you have a set of jumpers for your breadboard, these aren't necessary.

Multimeters

These devices give you the ability to check a range of different things, including the voltage, current, and resistance. It you're having problems with a circuit, they're invaluable tools to help you find out what's wrong. Without one, it's hard to tell if a particular connection is conducting well, or if a component is broken. They're also a lazy way of checking the value of a resistor (discussed later in this chapter).

Soldering Irons

Soldering irons are for creating permanent connections between two components or between one component and a circuit board. You'll need one if you're using stripboard, or if you buy a Raspberry Pi add-on that needs soldering together. We won't cover soldering in this chapter, but if you need to do it, there's an excellent guide called "Soldering Is Easy" at http://mightyohm.com/files/soldercomic/FullSolderComic_EN.pdf.

All of these tools are shown in Figure 11-4.

Figure 11-4: A set of tools for building your own hardware. None of them is essential for the projects in this chapter, though.

Hardware Needed for this Chapter

We've tried hard to keep the hardware for this chapter as simple, as easy to get, and as cheap as possible. In order to follow along, you'll need at least the following:

- Light emitting diode (LED)
- Resistors: 220 ohm, 1.1K ohm, and 6.2K ohm
- Solderless breadboard (any size should do)
- Jumpers for the breadboard (or single-core wire and a wire cutter)

- A way of connecting the breadboard to the Pi — either female-to-male jumpers or a Pi Cobbler

- MCP3008 chip
- Push switch
- Light-dependent resistor (LDR)

Each of these is discussed in the following sections.

The First Circuit

Before getting too far into the details of circuitry, let's create a simple circuit. You'll need a breadboard, an LED, a 220 ohm resistor, and some way of

connecting the GPIOs to the breadboard (either female-male jumpers or a Pi Cobbler).

This circuit is simply going to let you turn an LED (a type of light) on and off from Python. LEDs are a bit like mini light bulbs except for two things: firstly, they're much more powerefficient, so they shouldn't get too warm when running normally, and secondly they will only run one way round. That is, they have a positive leg and a negative leg, and the positive has to be connected to the positive and vice versa. The base of the LED should be round with a small flat section on one side. The flat side is next to the negative leg.

The resistor is there to stop too much power flowing through the circuit. As a general rule, you should always have at least one resistor of at least 220 ohms in a circuit; otherwise, you risk damaging your Pi and the other components. We'll look at this in a bit more detail later.

In this circuit, the resistor doesn't have to be 220 ohms; anywhere between 220 and 470 should be fine (you can try it with higher values, but the LED will be dim).

Resistors are colour-coded so you can tell their values. There are typically four or five bands of colour (a 220 ohm one will usually have four) and this should be red, red, black, and then silver or gold. Again, we'll discuss what this means a bit later.

The resistor and LED should be connected on the breadboard, as shown in Figure 11-5. The positive leg of the resistor (that is, the one not next to the flat side) should be the one that connects to the resistor.

R1 220 ±5%

1 51015 20
A
B
C
D
E

LED1 F Red (633nm)

G
H

I
J

1 5101520

Figure 11-5: Two diagrams for the circuit. The left one shows how to physically connect it while the right one shows how it’s linked together.

In order to make sure the circuit works, connect the wire from the resistor to one of the 3.3v pins on the Raspberry Pi (see Figure 11-6), while the lead coming from the LED should go to a ground pin. If it’s connected properly, the LED should light up.

SD
Card

3.3v 5v
*1 5v
*2 Ground

GPIO 4 GPIO 14
Ground GPIO 15
GPIO 17 GPIO 18
*3 Ground GPIO 23 GPIO 23
3.3v GPIO 24
GPIO 10 Ground GPIO 9 GPIO 25
GPIO 11 GPIO 8 Ground GPIO 7

*1 -- GPIO 0 on Revision 1 board GPIO 2 on Revision 2 boards *2 -- GPIO 1 on Revision 2 boards GPIO 3 on Revision 3 boards *3 -- GPIO 21 on Revision 1 boards
GPIO 26 on Revision 2 boards
Figure 11-6: Raspberry Pi pin layout. Don’t try to work out a logic for the pin numbering; there is none.

This, though, is just using the Raspberry Pi as a power source. In order to be able to control the circuit from Python, you first need to install the RPi.GPIO module. First make sure you have pip (a tool to help you access modules) installed with the following command (in LXTerminal, not in Python):

sudo apt-get install python3-pip

Then get the library with this command (also in LXTerminal):
sudo pip-3.2 install RPi.GPIO

When you’re working with the GPIO pins, you have to access the Raspberry Pi at a low level. Because of this, you can’t run the Python scripts normally. Instead, you need to run them with superuser permissions. This sounds fancy, but in fact it just means prefixing commands withsudo. So, for example, if you want to run a script in LXTerminal, you need to run:

sudo python3 your-script.py
Alternatively, you can start a Python shell with:
sudo python3
Or you can start IDLE 3 with superuser permissions by running the following in LXTerminal:
sudo idle3

Once you’ve installed RPi.GPIO, you just need to connect the circuit to a one of the GPIO pins. Disconnect the pin from 3.3v and connect it to pin 22. Once this is done, open a Python session (don’t forget sudo!) and enter the following:

>>> import RPi.GPIO as GPIO >>> GPIO.setmode(GPIO.BCM) >>> GPIO.setup(22, GPIO.OUT) >>> GPIO.output(22, True) >>> GPIO.output(22, False) >>> GPIO.output(22, True) >>> GPIO.output(22, False)

As you can see, setting the pin to True turns the LED on, while False turns it off. Figure 11-7 shows the running circuit.

Figure 11-7: The fully connected circuit on a small, solderless breadboard.

About Circuits

In essence, a circuit contains three things — a source of power, something that does something, and a place for the power to go (that is, a ground). Circuits always have to have all three. Nothing will happen if you connect a power source, but no ground. If you just connect a power source to the ground, then you have what's called a short circuit and it can draw a very large current and damager the power supply (see the section "Protecting Your Pi").

Circuits can vary from the very simple (like the one you made here) to the hugely complicated (such as a computer), but they all follow the same principles. In this chapter, we only have space to talk about the basics of using the Raspberry Pi's inputs and outputs, but if you're interested in circuits, you can take it as far as you want to go. If you want to take this further, the Penguin Tutor website has a good course to help you understand a little more about

what’s going on: www.penguintutor.com/electronics/.

Protecting Your Pi

In general, it’s very hard to physically damage a computer by programming. You might be able to corrupt some files (although even this is rare), but generally, no matter how much you mess things up, simply reinstalling the operating system will sort things out. However, when you’re adding things to the GPIO, you’re sending power directly to the CPU, and you can damage it. There are two properties of electricity that can cause problems, voltage and current.

With voltage, the rule is simple and very important. NEVER CONNECT MORE THAN 3.3 VOLTS TO A GPIO PIN. That’s very important, which is why we’re shouting. Quite a lot of hobby electronics components are designed to run at 5v because other processors run at that voltage level. However, if you connect these directly to the Pi, it can cause irreparable damage to the board. The result is known as bricking the Pi because afterwards its only use is as a brick.

CAUTION Never connect more than 3.3 volts to a GPIO pin!

You’ll notice that the Raspberry Pi has a 5v pin. This is only there to power external circuits that don’t come back to the Pi. If you need to connect a 5v device to the GPIO ports, you’ll need a logic-level converter. These are available for a few pounds and convert 5v signals into 3.3v and vice versa.

Whereas voltage can be thought of as the amount of energy electricity has, current is the amount of it that’s flowing through the wires. The two are connected by Ohm’s law, which states:

Voltage = Current × Resistance
Or, to put it another way:
Current = Voltage / Resistance
Current is measured in amps, voltage in volts, and resistance in ohms. In the previous circuit, there were 3.3 volts and 220 ohms, so that meant:
Current = 3.3 / 220 = 0.015A or 15mA

When using a Raspberry Pi, you must never draw more than 16mA from any one pin, or 50mA from all the GPIO pins combined. That means that you can light up only three LEDs at a time (or use more resistance to decrease the amount of current each one draws). It also means that you should never connect a GPIO pin

to a circuit unless there is at least 220 ohms of resistance in it. If you're ever unsure about resistors, always err on the side of caution and use larger ones than you have to.
Technically, this isn't actually correct because LEDs are a little different from many other components. This circuit will draw less power than that. However, unless you understand voltage drop on LEDs, it's best to stick with these guidelines.

If you need to draw more current, or just want to protect your Pi in case you accidentally draw more current, you can use an expansion board that has buffered input/output ports such as the PiFace, or alternatively, use a buffer-integrated circuit (IC or chip) to protect the GPIO ports.

Power Limits

The amount of current your Pi can supply is also limited by the amount of current it can get. Some power supplies will struggle to deliver much power, especially if there are several other things attached to the Pi like optical mice and USB memory. If you find that your Raspberry Pi becomes unstable or starts turning itself off when you're using GPIOs, then insufficient power may be the problem.

To combat this problem, you can upgrade to a power supply that can provide more current, or reduce the amount of current your Pi draws. Things like reducing overclocking in raspiconfig and removing non-essential peripherals will help.

Getting Input

In the previous example, you used the GPIOs to turn an LED on and off. This was the output side of GPIO — now it's time to look at the input. You'll also use the previous circuit, so leave it connected, but add a button.

The push button switch is a really simple component. When it's open (that is, not pressed) it doesn't connect the two pins, so no current can pass. When it's pressed, it connects the two pins and therefore acts just like a wire.

As you've seen, you'll need one resistor to stop too much current from flowing through the circuit. Just to be extra safe, we used a 1.1K ohm resistor here.

However, if you just connected a power supply to a resistor, to a button, to the GPIO, then when the switch is open, there won't be a circuit. If there's not a circuit, then the GPIO isn't on or off (or True orFalse, if you prefer). You need to ensure that there's a circuit that ties the GPIO pin to the ground (and therefore False) when the switch is open. This is known as a pull-down resistor because it pulls the GPIO down to 0v if there's nothing else connected to it. It needs to have quite a high value compared to the other resistor to ensure that enough of the electricity goes to the GPIO when the switch is closed. We used a 6.2K ohm resistor, but a 10K ohm one would work just as well.
Take a look at Figure 11-8 to see how to connect it. Note that the resistor between the GPIO and ground is the larger of the two. The button should, as the diagram shows, be connected between a 3.3v power source and GPIO 4.

15 10 15 20 A
B
C
D
E

F G H I J

15 10 15 20
R2
1k LED1±5% Red (635nm)

R1
220 ±5%

R3
6.8k ±5%

Figure 11-8: A circuit diagram that should help you understand what's going on.

Once it's set up, the following code will create a simple reactions game. It'll wait a random amount of time before turning the LED on. Then you have to press the button as soon as you can and it'll then tell you your reaction time. It's on the website as chapter11-reaction.py. Figure 11-9 shows the game in action.

Figure 11-9: You can expand this simple circuit to fit a wide range of projects.

```
import RPi.GPIO as GPIO import time
import random
from datetime import datetime

GPIO.setmode(GPIO.BCM) GPIO.setup(22, GPIO.OUT) GPIO.setup(4,
GPIO.IN)

GPIO.output(22, False) random.seed()

while True:
time.sleep(random.random()*10)
start = datetime.now()
GPIO.output(22, True)
while not GPIO.input(4):

pass
print("Your reaction time: ",

(datetime.now() - start).total_seconds()) print("Get ready to try again.")
GPIO.output(22, False)
```

Expanding the GPIO Options with I2C, SPI, and Serial

When you’re connecting two computers together, you can use an Ethernet network. This is a series of standards that define things such as the physical

cable you use, and the way your computer addresses other computers on the network. As long as all the computers are compatible with Ethernet, they should all be able to talk to each other.

There are also communications protocols designed for working with smaller pieces of hardware, such as different chips. There are three main ones we'll look at in this chapter—SPI, I2C, and Serial.

The SPI Communications Protocol

SPI or Serial Peripheral Interface uses four wires to provide a two-directional communication channel between two or more devices—a master (usually the Raspberry Pi) and one or more slaves (typically chips). The four wires include a clock wire that keeps everything in time, the Master Out Slave In (MOSI), Master In Slave Out (MISO), and a Slave Select (SS). Simply connect the pins from the Raspberry Pi to the corresponding pins on the slave, and you should be ready to go.

There are a wide range of expansion options for SPI, for example, analogue input.

Raspberry Pis have a range of inputs and outputs, but they're all digital. That is, they can only read on or off. That's fine for some things, such as buttons and controlling LEDs, but sometimes you'll need to read or write data on a scale. For example, you might want to read data from a sensor such as a light or temperature sensor. These don't give on-or-off values, but a range.

Values like this (that fall within a range) are known as analogue values (as opposed to digital). In order to read them, you'll need an analogue-to-digital converter (ADC). For this chapter, we're using an MCP3008, which is a chip that provides eight analogue channels and communicates with the Raspberry Pi using SPI.

Figure 11-10 shows what the pins on the MCP3008 do, and Figure 11-11 shows how to connect the circuit.
Analogue channel 0
MCP3008
Analogue channel 7

Power in

Reference voltage Analogue ground Clock
Data out
Data in
Slave select
Digital ground

Figure 11-10: The eight pins on the left input 0 to 7, while the pins on the right are to control the chip. This diagram is based on looking down from the top. You should see a semicircle cut into the plastic case. This should match the one in the diagram.

15 10 15 20
A
B
C
D

116
E IC
215

F
G
H 314
I
J

15 10 15 20 413
512
611
710
89

R2 220 LDR ±5%

Figure 11-11: How to wire the circuit. You can change which GPIO pins the pins on the MCP3008 connect to by altering the appropriate values in the code. The code for it is as follows. You'll find it on the website as chapter11-spiadc.py. Figure 11-12 shows the connected circuit.

Figure 11-12: The MCP3008 being used to convert the analogue signal from the LDR to a digital signal for the Raspberry Pi.

```
import time
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

# read SPI data from MCP3008 chip, 8 possible adc's (0 thru 7) def
readadc(adcnum, clockpin, mosipin, misopin, cspin): if ((adcnum > 7) or
(adcnum < 0)): return -1
GPIO.output(cspin, True)
GPIO.output(clockpin, False) # start clock low GPIO.output(cspin, False) #
bring CS low

commandout = adcnum
commandout |= 0x18 # start bit + single-ended bit commandout <<= 3 # we only
need to send 5 bits here for i in range(5):

if (commandout & 0x80):

GPIO.output(mosipin, True)
else:
GPIO.output(mosipin, False) commandout <<= 1
GPIO.output(clockpin, True)
GPIO.output(clockpin, False)
```

```
adcout = 0
# read in one empty bit, one null bit and 10 ADC bits for i in range(12):

GPIO.output(clockpin, True)
GPIO.output(clockpin, False)
adcout <<= 1
if (GPIO.input(misopin)):

adcout |= 0x1
GPIO.output(cspin, True)
adcout >>= 1 # first bit is 'null' so drop it return adcout

SPICLK = 4 SPIMISO = 17 SPIMOSI = 18 SPICS = 23

# set up the SPI interface pins GPIO.setup(SPIMOSI, GPIO.OUT)
GPIO.setup(SPIMISO, GPIO.IN) GPIO.setup(SPICLK, GPIO.OUT)
GPIO.set'up(SPICS, GPIO.OUT)

ldr_adc = 0; last_read = 0
tolerance = 5

while True:
# we'll assume that the light didn't change input_changed = False
# read the analog pin
ldr_value = readadc(ldr_adc, SPICLK,

SPIMOSI, SPIMISO, SPICS) ldr_movement = abs(ldr_value - last_read)
if ( ldr_movement > tolerance ): input_changed = True

if ( input_changed ):
print('Light = ', int(ldr_value)) last_read = ldr_value

# hang out and do nothing for a half second time.sleep(0.5)
```

Don’t worry too much about how the readadc() function works. It manipulates the individual bytes and sends and receives them down the MOSI and MISO wires. You can simply take it and use it to read the other ports on the ADC if you want to add more. Support for SPI is planned for the RPi.GPIO library, but at the time of writing, hadn’t been implemented. You can see how the project’s progressing at http://code.google.com/p/raspberrygpio-python/.

The main loop then uses this function to get the value of the ADC connected to the LDR and (if it's changed by more than the threshold), print its value onto the screen.

The I2C Communications Protocol

Inter-Integrated Circuit (I2C or I^2C, pronounced "I squared C") is a more powerful protocol than SPI. It still uses four wires, but one of them is the power line, and another is the ground, so there are only two data wires. Up to 127 devices can be connected to an I2C bus, and it has addressing capabilities rather than the rather simplistic slave select on SPI.

Quick2wire makes a series of I2C boards that expand the functionality of the Pi. You don't need these boards to use I2C, although they do make the process a bit easier. They also make a Python module that can communicate over I2C regardless of whether you're using their boards. You can find it at https://github.com/quick2wire/quick2wire-python-api.

Like SPI, support for I2C is planned for RPi.GPIO. Again, follow the website for up-to-date information.

The Serial Communications Protocol

The previous two methods are generally used for sending binary data between devices. Serial communication, on the other hand, is generally used for sending text back and forwards (although there are exceptions in both cases).

Serial communications are supported by the pyserial module that you can get with the following command:
sudo pip install pyserial

Once the pyserial module is installed, you just have to create a serial connection, then you can use write() and read() methods to send and receive data. For example, the following code will send the message "Hello" out of the serial port. It was created to work with a Ciseco PiLite, which will then scroll the letters across the screen.

```
>>> import serial
>>> pilite = serial.Serial("/dev/ttyAMA0", baudrate=9600) >>> pilite.open()
>>> pilite.write(bytes("Hello", "utf-8"))
```

Taking the Example Further

If you feel like taking this example further, try the following ideas.

Arduino

If you're interested in hobby electronics, perhaps the most useful kit is an Arduino. These are microcontroller boards slightly larger than Raspberry Pis. They have far more GPIOs (the exact number depends on the model), as well as analogue inputs and in some cases analogue outputs as well.

Perhaps the most useful thing about Arduinos, though, is the number of expansion boards (known as shields) that can slot onto them. With these you can quickly create powerful hardware with little (if any) wiring.

It is perfectly possible to connect your Raspberry Pi GPIOs to an Arduino to get them to talk via I2C or SPI, although most Arduino models run at 5v, so you'll need to use logic-level converters. They can also communicate over the USB using a serial connection. Some people consider Arduinos to be overkill when connected to a Pi, and it's true that almost anything you can do with an Arduino, you can also do with a Pi. However, the sheer amount of existing Arduino hardware and code makes them very useful companions to Raspberry Pis.

Perhaps the biggest drawback for readers of this book is that they're programmed in a dialect C++ rather than Python. The language has a different layout, but it's based on the same general principles. If you've read this far into the book, you shouldn't have too much difficulty picking up basic C++, although we won't deal with it here.

PiFace

The PiFace is an expansion board that slots straight onto the GPIO pins of the Raspberry Pi and is a really useful addition to the GPIO. As well as buffering the input and output (and so protecting your Pi and providing up to 500mA of current), it also has a pair of relays that can be used to drive higher powered components such as motors.

It's the same size as the Raspberry Pi, and fits neatly over the top. It's a great choice for getting started with simple robotics. It's around £25 ($41) so, while not as cheap as getting the components yourself, it's a great value for its ease of

use and the range of projects it supports. Figure 11-13 shows the PiFace connected to a “Model A Pi.”

Figure 11-13: The buffered outputs of the PiFace mean you can control far more without worrying about the small current limits on the Raspberry Pi.

Gertboard

The Gertboard is the Goliath of the Raspberry Pi GPIO options. It crams just about everything you could want into a board. It has the similar ATMega chips to Arduinos for learning microprocessing, a motor controller, digital-to-analogue converters, analogue-to-digital converters, push buttons, and a whole bunch of LEDs. Of course, all this comes at a cost, not just in terms of price (expect to pay a little under £60/$98), but also in terms of size, which could make it unsuitable for many projects where this is important (such as with robotics).

It is perhaps best thought of as a board for learning about electronics and control, and once you have a good idea what’s going on, you can design a smaller circuit for implementing your project.

The fact that it’s designed by the Raspberry Pi Foundation’s hardware supremo Gert Van Loo means you can trust that it’s been built by someone who really knows the Raspberry Pi inside and out (literally).

Wireless Inventor’s Kit

Everything we’ve done here so far has used wires to connect various components. For most projects, this is fine. However, some projects require a bit

more freedom. Ciseco has put together a Wireless Inventor's Kit, which contains everything you need to get started with simple radio communications and the Raspberry Pi.

It uses a radio system that's designed to work at a lower level than WiFi networking, and it's best used to connect sensors to your Pi remotely, or to use your Pi to control circuits from a distance.

Trying Some Popular Projects

You can create almost anything with a handful of components and a Raspberry Pi, but to get you started, the following sections include a few ideas.

Robots

These can range from simple wheeled devices powered by two motors, to vastly complex humanoid walking robots, to the bizarre such as those that move like snakes or spiders. To work with robots, you'll need to learn how to control motors (the GPIOs don't provide enough power to do this straight from the pin). The PiFace has a pair of relays, which is one option for getting started.

You'll also probably need some servos. Servos are a bit like motors, but can be turned a few degrees at a time. There are also a vast array of sensors that can be hooked up to your Pi to help your creation see the world around them.

Home Automation

Imagine a world where you can control your heating and lights from your smartphone. It's not an impossible dream—it can be achieved with a Raspberry Pi, some circuitry, and Python. Chapter 7 showed you how to control a Python program from the web, so combined with what you've learned here, you should be able to make this dream a reality.

Burglar Alarms

Maybe these are less glamorous than home automation, but they might keep you safe. With a Raspberry Pi's camera and a few things like passive infrared (PIR) movement sensors, you should be able to create your very own Fort Knox.

Digital Art

Art doesn't have to be lifeless painting in dusty rooms. Nor does it have to be sculptures or poetry or dance. It can be anything you want it to be. One of the emerging forms is digital art, which uses some form of computing to add a new dimension to an installation. It could be, for example, to add shifting light patterns, or pressure sensitive pads to create feedback for people touching it. You can let your imagination run wild, and then use a Raspberry Pi to give life to your imagination's creation.

Summary

After reading this chapter, you should understand the following a bit better:

- The pins that stick up from one corner of the Raspberry Pi are General Purpose Input and Outputs (GPIOs) that can be programmed from within Python.
- TheRPi.GPIO library offers a simple interface for switching the pins on and off, or reading values from them.
- You have to be careful when working with GPIOs because sending too much voltage, or drawing too much current, can cause irreparable damage to your Pi.
- You can expand your Pi by adding devices using protocols such as SPI, I2C, and Serial, all of which can be programmed from Python.
- One example of this is the MCP3008 chip, which has eight analogue-to-digital converters that can be read via SPI.
- Your Pi can control circuits of almost any complexity. The only limit is your imagination.

Chapter 12
Testing and Debugging

THERE ARE ALWAYS times when your code won't do what it should be doing. You see the inputs and work through the code, but somehow, it spits out an output that just shouldn't be possible. This can be one of the most infuriating parts of programming.

Investigating Bugs by Printing Out the Values

There are loads of ways to find out exactly what's happening, but one of the simplest is judicious use of print() statements. By using these to print out the value of every variable, you can usually get to the bottom of what's going on.

Take a look at the following code for a simple menu system. It doesn’t produce any errors, but whatever input you give it, it always says "Unknown choice". (It’s on the website as chapter12-debug.py):

```
choices = {1:"Start", 2:"Edit", 3:"Quit"} for key, value in choices.items():
print("Press ", key, " to ", value)
user_input = input("Enter choice: ")
if user_input in choices.values():
print("You chose", choices[user_input]) else:
print("Unknown choice")
```

Perhaps you’ve seen the problem already, but if you haven’t, what’s the best way to find it? The problem is that the if statement isn’t correctly identifying when the user_input is valid, so add some print() statements to see what’s happening:

```
choices = {1:"Start", 2:"Edit", 3:"Quit"}
for key, value in choices.items():
print("Press ", key, " to ", value)
user_input = input("Enter choice: ")
print("user_input: ", user_input)
print("choices: ", choices)
print("choices.values(): ", choices.values())
if user_input in choices.values():
print("You chose", choices[user_input])
else:
print("Unknown choice")
```

Straight away you should see the problem: choices.values() should be choices. keys(). The code was checking the wrong part of the choices dictionary. Make the change in the code and try running it again. With that bug fixed, everything should be fine. Oh no, it still doesn’t work! There must be another bug. Have another look at the output from theprint statements:

```
user_input: 1
choices: {1: 'Start', 2: 'Edit', 3: 'Quit'}
choices.values(): dict_values(['Start', 'Edit', 'Quit'])
```

Can you see why it’s failing? After the value of variable, the second most important thing is the data type of that value, so expand the print statements to

include more details about what’s going on there:

```
print("user_input: ", user_input)
print("choices: ", choices)
print("choices.values(): ", choices.values())
print("type(user_input): ", type(user_input))
for key in choices.keys():
print("type(key): ", type(key), "key: ", key)
```

If you run this, it should output:

```
Press 1 to Start
Press 2 to Edit
Press 3 to Quit
Enter choice: 1
user_input: 1
choices: {1: 'Start', 2: 'Edit', 3: 'Quit'}
choices.values(): dict_values(['Start', 'Edit', 'Quit'])
type(user_input): <class 'str'> type(key): <class 'int'> key: 1
type(key): <class 'int'> key: 2
type(key): <class 'int'> key: 3
Unknown choice
```

Now you can see that the cause of the problem is that user_input is a string, but the keys of choices are integers. The easiest way to solve this is to change the invocation of choices to make the keys strings:

```
choices = {"1":"Start", "2":"Edit", "3":"Quit"}
```

Now the basic logic of the program is working as expected; however, it still spits out loads of extra text that the user doesn’t want to see. Obviously you could just delete the print() statements, but they may be useful again in the future. You can keep them in the code, but have a flag that can be set to turn them off and on like as follows:

```
debug = True
choices = {"1":"Start", "2":"Edit", "3":"Quit"}
for key, value in choices.items():
print("Press ", key, " to ", value)
user_input = input("Enter choice: ")
if debug:
```

```
print("DEBUG user_input: ", user_input)
print("DEBUG choices: ", choices)
print("DEBUG choices.values(): ", choices.values()) print("DEBUG
type(user_input): ", type(user_input))
for key in choices.keys():
print("DEBUG type(key): ", type(key), "key: ", key)
if user_input in choices.keys():
print("You chose", choices[user_input])
else:
print("Unknown choice")
```

If you encounter any problems in the future, all you need to do is change the debug variable toTrue. Prefixing all the lines with DEBUG also makes it easy to see which output is normal, and which is debugging.

Finding Bugs by Testing

Debugging is the process of getting rid of problems in your programs. It can be quite challenging, but it can be even more difficult to find the problems in the first place. This might sound silly, but it’s true. As a program gets larger, the number of different ways it can be used increases, and the more different ways something can be used, the more places there are to check for bugs.

Imagine, for example, a word processor that has options for style, page layouts, file formats, layout managers, and so on. Bugs could lurk in any of these areas, so it’s important for the developers to check to make sure everything’s running as it should. They could even hide in combinations; for example, a problem may occur only if a particular font is used with a particular layout.

Checking Bits of Code with Unit Tests

The most basic form of testing programs is the unit test. This is where you take one small piece of code and make sure it’s behaving as it should. Typically, these are used to check that individual methods and functions are working properly.

Essentially, all a unit test does is run a piece of code with a particular set of inputs and check that their outputs are correct.
Take, for example, a function that takes a string of characters and returns a string with the same letters, but converted to uppercase. This could be implemented

and tested as follows:

```
def capitalise(input_string):
output_string = ""
for character in input_string:
if character.isupper():
output_string = output_string + character
else:
output_string = output_string + chr(ord(character)-32)
return output_string
print(capitalise("helloWorld"))
```

This should behave as expected. It works because the UTF-8 character encoding that Python uses stores characters as numbers, and uppercase letters are 32 places below their lowercase counterparts.

In this example, we've used a simple test case, and we're printing it to the screen to check manually. We can get Python to check the test case for us using the unittest module in the following code:

```
import unittest
def capitalise(input_string):
output_string = ""
for character in input_string:
if character.isupper(): output_string = output_string + character
else:
output_string = output_string + chr(ord(character)-32)
return output_string
class Tests(unittest.TestCase):
def test_1(self):
self.assertEqual("HELLOWORLD", capitalise("helloWorld"))
if _ _name_ _ == '_ _main_ _':
unittest.main()
```

This does more or less what the previous code did. If you run it, it'll check one string to make sure "helloWorld" goes to "HELLOWORLD". At this level, it's not much better or worse than just having a print statement.

When you run unittest.main(), Python runs every method in subclasses of

unittest. TestCase that start with test_. In this case it’s just test_1. The real advantage of using unit tests is that you can combine lots of tests in order to check at a glance whether things have worked properly.

You can add a second test case that checks that the string "hello world" capitalises to "HELLO WORLD":

```
def test_2(self):
self.assertEqual("HELLO WORLD", capitalise("Hello World"))
```

If you run this, you should get the following output:

```
FAIL: test_2 (_ _main_ _.Tests)
----------------------------------------------------------------Traceback (most recent call last):
File "capitalise.py", line 17, in test_2
self.assertEqual("HELLO WORLD", capitalise("Hello World"))
AssertionError: 'HELLO WORLD' != 'HELLO\x00WORLD'
- HELLO WORLD
? ^
+ HELLOWORLD
? ^
```

Oh dear, it looks like the test failed. You can see that the space wasn’t properly dealt with. If you go back to the original code, you can see that the problem is that everything that isn’t an uppercase character gets 32 taken off its UTF-8 value. Since space isn’t uppercase, this happens to it too, but this isn’t what the program should do.

Test cases should be designed to try a wide range of valid inputs. For example:

```
class Tests(unittest.TestCase):
def test_1(self):
self.assertEqual("HELLOWORLD", capitalise("helloWorld"))
def test_2(self):
self.assertEqual("HELLO WORLD", capitalise("Hello World"))
def test_3(self):
self.assertEqual('!"£$%^&*()_+-=',
capitalise('!"£$%^&*()_+-=')) def test_4(self):
self.assertEqual("1234567890", capitalise("1234567890"))
def test_5(self):
self.assertEqual("HELLO WORLD", capitalise("HELLO WORLD"))
def test_6(self):
```

```
self.assertEqual("`¬#~'@;:,.<>/?",
capitalise("`¬#~'@;:,.<>/?"))
```

If you run these, you'll see that most fail. The problem is a flaw in the program logic. The code leaves it alone if it's an uppercase letter and changes it otherwise. However, what you want is for the code to change it if it's a lowercase letter, and leave it alone otherwise.

If you change the capitalise function to the following, it'll do this:

```
def capitalise(input_string):
output_string = ""
for character in input_string:
if character.islower():
output_string = output_string + chr(ord(character)-32)
else:
output_string = output_string + character
return output_string
```

Now if you run the code, you should find that it passes all the tests.

By default, unittest will give you details only if one or more tests fail. Otherwise, it just returns an overall OK. For most purposes, this is what you want, but you can specify how verbose you want the output to be in two ways. If you're running the script from the command line, you can add the -v flag for more output. So, for example, if you've saved the program as capitalise.py, you can run the tests with verbose output using:

```
python3 capitalise.py -v
```

Alternatively, you can specify that you want a more verbose output in the code itself by changing:

```
if _ _name_ _ == '_ _main_ _':
unittest.main()
```

to:

```
if _ _name_ _ == '_ _main_ _':
unittest.main(verbosity=2)
```

Before going any further, we should point out that the capitalise() function is here for this example. If you actually need to capitalise text, you should use upper() method of the string class. For example:

```
>>> 'hello world'.upper()
```

Getting More Assertive

All these tests have a call to self.assertEqual(). This line tells the unit test module what the output of the test should be. That is, the test should pass if the two values passed are the same, and fail if they're different. This covers a large proportion of cases, but you may wish to check different things. There are a number of different assert methods that you can use in your tests.
These check that various structures are the same:

■ assertSequencesEqual(sequence1, sequence2) ■ assertListEqual(list1, list2) ■ assertTupleEqual(tuple1, tuple2) ■ assertSetEqual(set1, set2) ■ assertDictEqual(dict1, dict2)

With these structures, you may want to check that the value is in the structure rather than if two structures are the same. The following methods check if a value is in a structure:
■ assertIn(value, structure) ■ assertNotIn(value, structure)
Strings are a special type of structure and have their own method:
■ assertMultiLineEqual(string1, string2)
You can also check values using tests other than equality using these assert methods:

■ assertNotEqual(value1, value2) ■ assertGreater(value1, value2) ■ assertGreaterEqual(value1, value2) ■ assertLess(value1, value2) ■ assertLessEqual(value1, value2)

There are also a few that allow a margin of error:
■ assertAlmostEqual(value1, value2) ■ assertNotAlmostEqual(value1, value2)
These check that the two values differ (or not) by less than 0.000001. These are useful if you're testing floating-point functions that might have small rounding errors that are acceptable. You can also test anything that you can reduce to a True orFalse value using:
■ assertTrue(value) ■ assertFalse(value)
Whatever you want to check, each test_ method should have oneassert method call that is used to determine the success or failure of that particular test.

You can use these unit tests in a number of ways. There is a style of development called testdriven development that says that the tests are the first thing you

should write and then you use those tests as the specifications for the code. Most programmers, though, write the tests towards the end of development to make sure everything's working properly.

Using Test Suites for Regression Testing

Writing programs isn't usually a single effort. You don't usually sit down, create software, and then stop and go on to do something else. Instead, you generally code some of the features, distribute it to users, then fix bugs and add new features in later versions.

There is a risk of breaking things that once worked as you add new features, so it's important to test not only newly created things, but also older things that have worked. Testing older code is called regression testing, and having a properly ordered set of tests makes it really easy.

To make sure that you're not introducing bugs into previously working code, you should rerun the tests after you make any changes. However, as your programs become bigger, you'll end up with more and more tests. Eventually, you'll get to the point where it's not practical to run every test every time. You can group tests together into test suites. These allow you to test just particular areas of your program at a time.

Using the previous code, you can change the final code block to:

```
if _ _name_ _ == '_ _main_ _':
letters_suite = unittest.TestSuite()
symbols_suite = unittest.TestSuite()
letters_suite.addTest(Tests("test_1"))
symbols_suite.addTest(Tests("test_2")) symbols_suite.addTest(Tests("test_3"))
symbols_suite.addTest(Tests("test_4"))
symbols_suite.addTest(Tests("test_5"))
symbols_suite.addTest(Tests("test_6"))
all_suite = unittest.TestSuite()
all_suite.addTest(letters_suite)
all_suite.addTest(symbols_suite)
unittest.TextTestRunner(verbosity=2).run(all_suite)
```

This bit of code by itself does exactly what the previous code block did. That is, it runs all the tests. However, it has grouped them into different test suites.

There’s letters_suite that runs the test that checks letters, symbols_suite that runs the tests that check symbols, and all_suite that combines both of them. You can use the final line to run any of these three suites.

Using this code, you should find it quite easy to build a simple testing menu to help you make sure that everything’s running smoothly.

Testing the Whole Package

Unit testing is great because you can automate it, and quickly check that everything’s running properly. However, it doesn’t cover everything. Even though everything seems to work properly by itself, you may still find that there are problems when everything comes together.

In commercial software development, after the unit tests have been done, the code will be passed to the quality assurance team to make sure everything’s working fine. This team will outline a series of test cases that cover how the software will be used. It should check every aspect of the program and test it with a variety of inputs to make sure it behaves as expected. This is sometimes done manually with testers interacting with the software just as users would, and sometimes by specialist testing software that can simulate mouse and keyboard input. Of course, it’s unlikely that you’ll have a quality assurance team to help you with your software, but there are some things you can take from the professional approach. You should be methodical. Before you start testing, make a list of everything that the program does, and come up with test input and expected outputs. You can then go through this list and make sure it’s all functioning correctly.

It’s probably a bit excessive to do this after every code change, but you should do it periodically, and especially before any big releases.

Making Sure Your Software’s Usable

By the time you’ve finished a program, you know everything there is to know about it. You know how to interact with it, how to get the best out of it, and what all the various options are. Your users, however, don’t have any of this knowledge. Your software has to help them understand it and provide enough information for them to know what to do. After all, it doesn’t matter how awesome your features are if the users don’t know how to invoke them.

User testing is the area of testing devoted to making sure this is possible. In an

ideal world, you'd get a room full of people, sit them down in front of your software, and ask them to perform certain tasks and see how they get on. Again, you're unlikely to be able to do this. Sometimes you may be able to persuade a friend or relative to help you out, but the more programs you create, the fewer volunteers you seem to find. The only real solution to this is to listen to people using the software, and make sure to ask for feedback.

How Much Should You Test?

There's an old saying about software bugs that goes, "Absence of proof isn't proof of absence." Basically, no matter how much you test your software, there's no way of ever proving that there aren't any bugs in it. In fact, it's almost impossible to write software that doesn't have any bugs in it. The purpose of testing isn't to make perfect software, but to make software that's good enough. What "good enough" means will vary from project to project. The more important the software, the more you should test it, but all software deserves at least some testing. It's not as glamorous as implementing new features, but most of the time it is more important to have a few features that are properly tested than have loads that are buggy, so it's worth spending some time writing unit tests and making sure everything's working properly. After all, it could well be your data that the program loses when there's a problem.

Summary

After reading this chapter, you should understand the following a bit better:

- Debugging is the process of removing any problems from the code.
- Judicious use of print() statements can help you find out what the problems are.

- It sometimes helps to have a way of switching these print() statements on and off so you can reuse them if you find more problems.
- Testing is the process of finding bugs in code.
- Unit tests are the most basic form of testing and can be automated using the unittest module.
- Test cases can be grouped together into test suites to help you test particular areas of a program.
- It's easy to accidentally introduce new bugs when you add features, so you should always regression test after you make changes to your code.

■ Unit tests won’t pick up all problems though, so you should also test at a completesystem level.
■ Usability problems are also bugs, so you need to listen to your users to make sure they are addressed.

This brings us to the end of the book. Hopefully, you’re now confident and knowledgeable enough to create your own programs. Don’t worry if you feel you don’t know everything about every aspect of Python, very few people do. If you ever get stuck, you can always refer back to this book, or the Python documentation at http://docs.python.org/3/.

Hopefully you’ve seen that programming isn’t overly complex, and if you break up a problem into small steps, it’s usually quite straightforward to code. The main thing to remember is that programming should be fun! Find an area that interests you and explore it. Despite its small size, there’s very little you can’t do with a Raspberry Pi.

Index

SYMBOLS

* (asterisk), 203
: (colon), 17
{} (curly braces), 37, 113, 114, 137 {{}} (double curly braces), 153 == (double equal sign), 19
“” (double quote marks), 30, 205
- (hyphen), 9
(number), 16
. (period), 203
+ (plus sign), 203
" (quote marks), 30, 31, 114, 205 () round brackets, 34
; (semicolon), 113, 137
[] (square brackets), 34
/// (three slashes), 73
<> (triangular brackets), 149

Application Programming Interface (API), 145, 159
apt-get command, 7, 10, 77
Arch, 7

Arduino, 236–237
arguments, 101–102, 154, 157, 158, 214
array_spec (vertex_attrib), 114
ArraySpec object, 114
array_spec.glsl (), 114
assert method call, 252
asterisk (*), 203
ATMega chips, 237
attribute shader, 137
attributes, 50
audio. See sound
audio Swiss Army knife (sox), 184, 196
AWB, 192

A

-a flag, 202
actions, as methods to run when event happens, 60

ADC (analogue-to-digital converter), 232, 237–238, 239
add_options () calls, 203
addStretch () method call, 67
all value, 157
--all, 202
all_suite, 253
Alpha (transparency) value, 111
Alt+Tab, to switch back to non-Minecraft window, 164
ambient light value, 122
analogue input/signal/values, 232, 233
analogue-to-digital converter (ADC), 232, 237–238, 239
animation, 86, 169
answering questions, by program, 181–182
apple_freq loops, 174
apple_in variable, 174

B

babbage, 15, 17, 49

Babbage, Charles (inventor of concept of computer), 15
babbage.color () method, 19

back button, in web browser, 65
background image, for platform game, 99
backups, 215
backup_types list, 213
Bash environment, 199
begin_fill () method, 18
bind (), 136
birthday () method, 50
BLEND_RGB_SUB flag, 187
blit (), 187
blit (draw), 186
block types 18/22/246, 168
Blue value, 111
bookmarks picker, 68
bool type, 31–32, 38, 42, 55
box layout, 63, 66, 74
boxsize variable, 24
breadboard, 220, 221, 223, 224 break statement, 41
bricking, 228
brightness variable, 120, 121 Browser class, 65, 66, 68, 69
Browser’s self.menu_bar, 65 BrowserWindow class, 71, 72 buffered input/output ports, 229 buffer-integrated circuit (IC), 229 buffers, 112
bumpedspere.py, 136
burglar alarms, 239
button clicks, events as, 60, 65
button presses, 74
buttons, 7, 59, 65, 66, 73, 74, 76, 103, 229, 230, 232, 238
bytes data type, 179

C

C language, GLSL as similar to, 113
C++ language, 237
camera module, 5, 184, 190–193
camera object, 186
cameras, 177, 184, 188
capital letters, 49–50
capitalise () function, 250
cascade file, 189

cat command, 199
cat/proc/cpuinfo, 200
cd command, 8, 165
Celsius (temperature), 148
Central Processing Unit (CPU), 109, 112, 199 chat server, building of, 141–144
checkbounds () function, 24
choices.keys (), 242
choices.values (), 242
Chrome, 68
chunk_size, 179
circle () method, 15
circlearea (1) function, 20
circuits, 224–227, 230, 232, 233
classes. See also specific classes
advantage of, 51
building objects with, 49–54
defined, 49, 55
inheritance, 58
instances of, 50
in Qt graphical toolkit, 59
client, 141
clock wire, 231
clock.tick (fps), 83
closing tag, 149
code, reusability of, 43–46, 51, 55
code blocks
indentation in, 38–39
loops as, 17
code editor, 21
code examples
adding light to spinning cube, 123–126 chat server, 142–143
circuit with MCP3008, 233–235
converting speech to text, 180
copying files into tar.gz file, 209–212 for creating browser window, 63–64
design of platform game, 79–82
as downloads, 21
for making game Snake, 169–172
for saving sound, 177–179
speech recognition oracle (Pyri), 182–184 spinning cube, 116–119

for spinning cube visualiser, 130–135 collided () method, 96
collide_get_y () method, 92
colliderect () method, 88
collisions, in platform game, 88–90, 96 colon (:), 17
color (colour1, colour2)
method, 18
color variable, 120
combo box/combo box changes, 68, 74 command line, 8, 9. See also Linux
command-line flags, 202–203
communications protocols
Inter-Integrated Circuit (I2C), 235–236, 239 serial, 236, 239
Serial Peripheral Interface (SPI), 231–235, 236, 239
comparison operators for numerical types, 29 complex layouts, 63
compressed archive, 212
computer vision features, adding of with OpenCV, 187–190
computers
communicating between two, 142, 144, 159 getting sound into, 177–181
as under-utilised, 1
conditional logic, flow of, 19
conditionals, 18–20
<condition>B, 19
conditions, as true or false, 19
connect calls, 60
connection variable, 141
controls, adding, 60–62
convert (), 100
cookies, 155, 160
coordinate systems, 107
copy.deepcopy (), 45
count (), 36
CPU (Central Processing Unit), 109, 112, 199
crashing, 6, 196
Creative Commons, 85
crontab feature, 215
Ctrl+C, 39, 41, 82
curly braces ({}), 37, 113, 114, 137
current, 228
cursor, as turtle, 20
cv2.Cascade Classifier (), 189

D

data types, 32, 243
database, 46–48, 51–54
datetime.datetime.

fromtimestamp (), 214 DEBUG prefix, 245
debugging, 241–245, 255
demos subfolder, 136
depth, creating sense of, 100
.dest="filename" flag, 203 detectMultiScale (), 189, 190 /dev/video0, 186
development files, 77
dialogs, defined, 72, 74
dictionaries

grouping values in, 55
non-sequential values in, 37–38 as odd data type, 40

dictionary of dictionaries, 48
Digia, 57
digital art, 239
digital inputs/outputs/signal, 232, 233 digital-to-analogue converter, 237
directory tree, 8
dirs subdirectories, 213
dmesg, use of to locate source of

problems, 12
Doom class, 93, 95, 96
dot product function, 122, 123 double curly braces ({{}}), 153 double equal sign (==), 19
double quote marks ("""), 30, 205 draw () method call, 137
drawing pictures, 13–21, 166–167 Dropbox, 208
DVI input, 6

E

-e jpg option, 192
electricity properties, 228
ElementTree, 182

elif, as short for else if, 19
elif (else-if) statements, 42
else statements, 42
end_fill () method, 18
equipment, required/optional, 5
error messages, 6, 201
escaped characters, in regular expressions, 206 /etc, 199
Ethernet network, 231
etho0, 140
exceptions, catching, 42–43
exercise 1, chapter 3, 33
exercise 1, chapter 4, 62
exercise 1, chapter 7, 149
exercise 2, chapter 3, 43
exercise 2, chapter 4, 73
exercise 2, chapter 7, 155
exercise 3, chapter 3, 48
expansion boards (shields), 236
extension .dev, 77
extension .py, 11
extra features, getting of from modules, 54–55 eye_matrix, 129

F

F1, 25
Fabric module, 207 factor_down value, 190 factor_up value, 190

False value, 31–32, 38, 39, 55, 88,
91, 92, 226, 229, 252
Fedora, 7
female to male jumper wires, 219–220, 223
file, cannot be used as variable name, 216
filename attribute, 203
filenames, 72, 213, 214
files, working with, 216–217
filter, 72
Fireball class, 99
fireball_high_speed global variable, 95
fireball_low_speed global variable, 95

fireball_plain, 96
fireballs, in platform game, 94–96
Firefox, 68
5v pin, 228
FLAC format, 180, 184
flac program, 184, 196
flag -h, 202, 203
flags, 187, 202–203, 205, 214, 250
flip (), 186
float, 120
float () function, 32
float type, 55
floating-points, 32, 120
for loops, 39–40, 55, 179, 187, 213, 216
forecast_dict data structure, 148
forms, 153–155, 160
forums, 7
forward button, in web browser, 65
forward () method, 15
4D vector, 115
fragment shader, 111, 113, 115, 116, 120
fragment_glsl variable, 111, 113
frame_rate variable, 127
frames, 127
freezing, how to counteract, 107
Fritzing, 222
from_byte () method, 127
functions. See also specific functions
defining, 24
making code reusable with, 43–46, 55 methods as defined much like, 50
starting with lowercase letters, 49
use of to structure code, 20–21
value of in allowing code to be reused, 25

G

games
cat and mouse, 21–25
distribution of, 99

Minecraft, 163
platform, 77–108
simple reactions, 230
Snake, 169–175

General Purpose Inputs and Outputs (GPIOs), 2, 219
Gertboard, 237–238
get method, 152
GET requests, 153, 154, 157
getBlock () method, 173
getOpenFileName () method, 72
github.com/ashtons/picam, 191
github.com/quick2wire/
quick2wire-python-api, 235
glesutils.Texture.from_surface () method, 136
gl_FragColor, 115, 120
GL_LINE_STRIP, 116
GL_POINTS, 116
gl_PointSize, 115
gl_Position variable, 114, 115, 129
GLSL (Graphics Library Shader Language), 113, 122, 137
GL_TRIANGLE_STRIP, 116
Go button, in web browser, 66
go_btn_clicked () method, 67
Google
Drive, 6, 208
home page, 64
request, 183
search, 69
speech-to-text web service, 180, 196
google.com, 140
GPIO options, expansion of, 231–236
GPIO pins, 219, 220, 224–226, 228, 229, 232, 236, 237, 239
GPIOs (General Purpose Inputs and
Outputs), 2, 219
graphical programming, 57–74
graphical system (LXDE), 7. See also
LXDE (Lightweight X11 Desktop
Environment)

graphical user interface (GUI) programming, 58–60, 128, 197
graphics cards, 109
Graphics Library Shader Language
(GLSL), 113, 122, 137
Graphics Processing Units (GPUs), 109, 111–112
gray_frame image, 189
Green value, 111
grid layout, 59, 63, 66, 74
grow_in list, 173
GTK graphical toolkit, 57
GUI, 197
gzipped file (compressed file), 215

H

Haar cascades, 188, 189, 190
hardware, interfacing with, 219–239
HDMI-to-DVI converter, 6
HDMI-to-VGA converter, 6
headers parameter, 180
“Hello World!” website, 151
HelloHandler class, 153, 154
hello-template.html, 153, 155
hello-template.html.

self.get_argument (), 154
help= parameters, 203
-help/–help flag, 9, 202, 203
hierarchy, 197
high-definition multimedia interface (HDMI)

video output, 6
hobby electronics, 236
home automation, 239
home directory, 9
/home, 199
/home/pi root directory (HTTP), 150, 153 /home/pi/images with mkdir/home/

pi/images , 193

horizontal boxes, 63
hostnames, 140
hosts, 139–140, 159
HTML (Hypertext Markup Language), 149–150,

152, 153, 158, 160
HTTP (Hypertext Transfer Protocol), 149, 150,
153, 157, 160, 180
http://, use of, 67, 70
HTTPError, 183
https://, use of, 70
http.server module, 151, 160 hyphen (-), 9

I

I2C (Inter-Integrated Circuit)
communications protocol, 235–236, 239
IC (buffer-integrated circuit), 229
icosa.py, 136
IDE (Integrated Development
Environment), 11, 21
identical code, repetition of, 16
IDLE 3, 11, 13, 18, 21, 226
id_rsa file, 208
id_rsa.pub file, 208
if block, 41, 42, 89, 90, 100
if clause, 20
if condition, 19
if .. elif .. else block, 18
if line, 205
if statements, 25, 31, 41–43
ifconfig, 140
ImageHandler class, 194
images
background image for platform game, 99 creation of training ones, 190
for games, 84
JPEGs, 192
manipulation of, 187–190
import copy, 45

import lines, 54
increment () function, 46
indentation, 38–39, 41, 114
index (), 36
indexes, need for, 37
index.html file, 150
indices_face_1 tuple, 111
information, storage of in text files, 216–217
inheritance, 51, 58, 84
initialise pygame, 83
_ _init__ method, 50, 51, 60, 87, 98
input, getting, 229–231
input () function, 43
instability, of Raspberry Pi, 229
instances, created/initiated, 50
int class, 127
int () function, 32
int type, 55
integer (int), 29, 32, 244
Integrated Development
Environment (IDE), 11, 21
Inter-Integrated Circuit (I2C) communications protocol, 235–236, 239
Internet, sending data over, 193
Internet Protocol (IP) address, 139, 140, 144
IPv4 (version 4, IP address), 139, 144
IPv6 (version 6, IP address), 139

J

JavaScript Object Notation (JSON), 147–149, 159, 181
JPEG image, 192
json module, 147
jump () method, 90, 99
jumpers, 223
jumping sound effect, 98

K

Kelvin (temperature), 148

kernel buffer, 7
keyboard, 184
keypress events, 58, 91, 96
keys, need for, 37
key/value pairs, in dictionaries, 37, 40

L

LANs (local area networks), 144 laplacian transform, 187
layouts

box layout, 63, 66, 74
complex layouts, 63
grid layout, 59, 63, 66, 74 QGridLayout, 59
QHBoxLayout, 63
Qt graphical toolkit, 59 QVBoxLayout, 63

LDR (light-dependent resistor),
223, 233
Leafpad, 11, 103
LED (light emitting diode), 223, 224–225, 228, 229, 230, 238
left () method, 15
length () function, 121
letters_suite, 253
LibreOffice’s Writer, 11
Lightweight X11 Desktop
Environment (LXDE), 7–8, 57, 197
lines image, 187
Linux
command line, 180, 188, 197–199 command-line book, 9
command-line flags, 202–203
command-line program scp, 207 command-line tools, 196
compressed archive, 212
crontab feature, 215
flexibility of, 2
kernel, 197
log files, 205
as open source, 85
output from running command on, 201

start of directories in, 8
system commands, 157
timestamp, 214
using Arch or Fedora, 7
list of lists, 35, 48
listen (), 141
lists, 33–34, 35–36, 37, 45, 55
live streams, 193–196
live.jpg, 194
local area networks (LANs), 144
local variables, 50, 84
localhost, 140, 150
local-only address, 144
lo.eth0, 140
logic-level converter, 228
login method, 159
login-fail.html, 158
login-template.html, 158
logout_template.html, 159
loops
for, 39–40, 55, 179, 187, 213, 216 apple_freq loops, 174
bool type used in conditions for, 31 in cat and mouse game, 25
defined, 16
nested, 40–41
as way to control how Python moves through program, 25
while, 38, 55, 173, 187, 189
lowercase letters, 49
ls, 202
LXDE (Lightweight X11 Desktop Environment), 7–8, 57, 197
LXTerminal application, 8–11

M

main function, 114, 115
MainHandler class, 152
master, 231
Master In Slave Out (MISO), 231
Master Out Slave In (MOSI), 231
matrices, 112

matrix multiplication, 129
mc object, 165
MCP3008 chip, 223, 232, 233, 239
/media, 199
/media/MyStick, 199
menus, adding, 62, 71, 74
metering mode, 192
method calls, 74
methods. See also specific methods

in babbage, 15
defined, 49
as defined much like functions, 50 starting with lowercase letters, 49 use of to structure code, 20–21
use of to take care of much of the work, 25

micro USB power supply, 6
microcontroller boards, 236
microphones, 177
microprocessing, 237
Midori, 68, 85, 139
Minecraft, 163–169

minecraft-pi-0.1.1.tar.gz , 164 MISO (Master In Slave Out), 231
mixer, initialising, 98
-m option, 214
modified time (--mtime option), 214 module_example import line, 54 modules. See also specific modules

advantages to creating, 55
getting extra features from, 54–55 importation of, 25
as time-savers for getting functionality, 66

monitors, 5, 6
MOSI (Master Out Slave In), 231
motor controller, 237
motors, 237, 238
mouse, 57, 184, 229
move (), 90

move_ip (), 90
movement variable, 173
move_x () method, 94
move_y (), 89
movies, making, 184–196
MP3 sound files, 98, 126
mpg123 command-line tool, 126
–mtime option (modified time), 214 multimedia

making movies, 184–196
using PyAudio to get sound into your computer, 177–184

multimeters, 222, 223
music
copyrights on, 99
downloading source, 127
for spinning cube, 126–136
mutable data types, 45
myfile.txt, 217
mylevel file, 101
MyStick, 199

N

name server, 140
namespace clashes, 55 nested loops, 40–41 network interfaces, 140 network port, on model B Raspberry Pi, 5 network value, 157
networking, scripting with, 207–209
Nokia, 57
non-sequential values, 37–38
NOOBS, 7
normal, as vector that sticks out of a face at 90

degrees, 121–122, 123, 136
normalize () function, 122
number (#), 16
numbers, 29–30
numerical operations, 29
numerical operators, 30

O

object recognition, 188
objects
building of with classes, 49–54
as mutable data type, 45
OGG files, 98
Ohm’s law, 228
open () function, 216, 217
open source, 84, 85
Open Web Application Security Project, 159 OpenCV, 196
openCV module, 187–190, 191
opencv_traincascades
program, 188
opengameart.org, 84, 97, 98
OpenGL, 109, 111, 112, 136
OpenGL ES, 120
OpenGL ES 2.0, 136
opening tag, 149
OpenWeatherMap.org, 147, 155
operations on lists, 35–36
operations on sets, 38
operators for numerical types, comparison, 29 OptionParser object, 203
#options, 86
optparse module, 202
os module, 209, 217
os.path.getmtime (), 214
os.path.join (), 214
os.path.splittext () function, 213 os.walk () function, 213 output.wav file, 179
overclocking, 10, 107, 190 owasp.org, 159

P

page IDs, 153
parallax scrolling, 100
parameters

defined, 44
in methods start with self, 50 optional ones, 46

use of brackets with, 60–61

Parent class, 50, 51
parser.add_option (), 203, 212 pass statement, 82
passiver infrared (PIR) movement

sensors, 239
passwords, 8, 207, 209
PCBs (Printed Circuit Boards), 222 Penguin Tutor website, 227
period (.), 203
peripherals, 6, 205, 229
permissions, superuser, 226
Person class, 50, 51
physics, realistic game, 103–108
Pi Cobbler, 220, 223
pi directory, 9
pictures, drawing of, 13–21, 166–167 PiFace, 229, 237, 238
pi.minecraft.net, 164
pin 22, 226
pin joint, 106
pip (tool), 224–225
PIR (passive infrared) movement

sensors, 239
platform game
adding movement to sprites, 86
building of, 79–82
creating a world, 86–88
detecting collisions, 88–90
drawing sprites for, 84–85
making a challenge, 93–97
making it your own, 97–103
moving left and right, 90–92
reaching the goal, 92–93
Player class, 83, 84, 86, 90, 94, 98, 99 player object, 165
player_image variable, 84
player_plain, 85, 96
player.setPos(x, y, z), 175 plus sign (+), 203
PNG files, 84, 100, 187, 192

pop (), 173
pop (x), 36
port 22, 140
port 80, 140, 150
port 8000, 150
port 8888, 152
ports, 140, 141
position_matrix, 129
POST requests, 153, 154, 160
postTo-Chat () method, 173
power
as most common cause of problems with Raspberry Pi, 6
with power comes complexity, 137
supply/limits, 5, 229
powered USB hub, 5, 6
prefixes
DEBUG prefix, 245
with module name, 55
protocol prefix, 67, 70, 73
sudo prefix, 11, 226
<pre>B tags, 158
print () function, 20, 43
print () statements, 201, 241, 242, 243,
244, 255
Printed Circuit Boards (PCBs), 222
print-file.py, 203
private key, 208
problems, common ones, 6
processes value, 157
/proc, 199
/proc/cpuinfo, 199
program flow, control of, 38–39
programmers/programming, 1
projects
burglar alarms, 239
digital art, 239
home automation, 239
robots, 238
protection, for Pi, 228–229

protocol prefix, 67, 70, 73
prototyping boards, 221
pseudo-widgets, 67
public address, 144
public key, 208
pull-down resistor, 229
push button switch, 223, 229, 238
PyAudio module, 177, 196
PyAudio object, 179
pygame directory, 78, 185
PyGame mixer, 127
PyGame module, 77, 82–86, 103, 106, 107,
110, 173, 185, 187, 188, 196
pygame.key.get_pressed (), 91 pygame.sprite.Sprite, 84
PyMunk module, 103, 106, 107, 108
Pyri (speech recognition oracle), 182
pyserial module, 236
pyside module, 57, 62, 73, 74, 77
Pythagorean theorem, 121
Python
bringing everything together, 46–48
building objects with classes, 49–54
controlling the way the program flows, 38–43
documentation for, 25, 255
getting extra features from modules, 54–55
making code reusable with functions, 43–46
shell, 11, 226
storing values in structures, 33–38
turtle module, 13–21
using of from saved programs, 11
using of from shell, 11
variables, values, and types, 27–33
ways to write programs for, 11
Python 2, 187
Python 3, 185, 187, 207
Python API, 164
Python interpreter, 11, 27, 30, 39, 165, 200 Python variable to print, 153

python3 client.py, 142 python3 server.py, 142

Q

q key, 188
QActions, 71
QColor type, 73
QColorDialog, 73
QColorDialog.getColor (), 73 QComboBox, 68, 69
QFileDialog, 72, 73
QGridLayout, 59
QHBoxLayout, 63
QLineEdit, 66
QLineEntry, 70
QMainWindow, 62, 64, 71, 74
QPlainTextEdit, 66
QPushButton, 70
QPushButtons, 65
QSlider, 70
Qt graphical toolkit, 57, 59, 62, 66, 67, 72,

73, 74
QTextEdit, 66
questions, asking of the program, 181–182 Quick2wire, 235
quote marks ("), 30, 31, 114, 205
QVBoxLayout, 63
QWebView, 63, 64, 65, 70, 71, 73
Qwidget, 59
raspi-config, 10, 107, 190 raspistill, 191, 192
reactions game, simple, 230
read () method, 236
realistic game physics, 103–108 record_sound () function, 179 Rect class, 78, 90
recv () method, 141
Red value, 111
red varying variable, 129
-regex flag, 205, 214
regression testing, 252–253, 255 regular expressions (regex), 203–207,

214, 217
relays, 237, 238
re.match (), 204, 207, 214

RenderPlain, 85, 96
re.search (), 204, 207, 214 reset () method, 95
resistors, 223, 224–225, 228, 229 RGB values, 73, 87
right () method, 15
robotics, 190, 237, 238
root directory, 8, 197–198, 213, 217 round brackets (), 34
RPiGL module, 110, 136
RPi.GPIO module, 224–225, 226,

235, 239
-r flag, 205, 214
Run button, 103

R

randint (a , b) method, 95
range (x, y), 39
Raspberry Pi

as great device on which to learn programming, 2
model A, 5
model B, 5
online shop, 6
speaking to, 180–181
turning of into mirror, 185–186 Raspberry Pi Foundation camera module,
184, 190–193, 196
Raspbian, 6, 7, 197, 199, 205

S

sample_rate , 179
scaling factor, 112
scenery, adding of, 99–101
scp (secure copy), 207, 208, 215, 217 Screen class, 49
screen_x variable, 95
scripting

bringing it all together, 209–215
command-line flags, 202–203
getting started with Linux command line,

197–199
networking with, 207–209
regular expressions (regex), 203–207 using subprocess module, 200–202
working with files in Python, 216–217

SD card, 5, 6, 196, 219
search bar/box, in web browser, 69, 70
security, 155–159, 196
self.assertEqual () call, 250
self.image variable, 84
self.programl.uniform.transform_ matrix.value attribute, 116
self.rect variable, 84
self.render (), 152
self.url_entry.text () parameter, 67 self.verteces_buffer.draw (), 116
self.webview.load call, 69
self.write (), 152
semicolon (;), 113, 137
send () method, 141
sequences, 33
serial communications protocol, 236
Serial Peripheral Interface (SPI) communications protocol, 231–235, 236, 239
server socket, 141
servers, 141, 160
servos, 238
setBlock method, 165, 166, 175
setBlocks (), 175
setMaximumSize () method, 66, 67 setMinimumSize () method, 67
sets
grouping values in, 55
as mutable data type, 45
non-sequential values in, 37–38
operations on, 38
setup.py, 103
shader objects, 111
shaders, 128, 136. See also attribute shader; fragment shader; vertex shader
shields (expansion boards), 236
simple reactions game, 230
size variable, 187
skeleton of the program, 79

Slave Select (SS), 231
slaves, 231
slider movements, 74
small_gray_frame image, 189
Snake (game), 169–175
SoC (System on a Chip), 109
socket module, 141
sockets, 140, 141, 159
software
installation of, 10–11
making sure it’s usable, 254
soldering, guide on, 222
soldering irons, 222, 223
solderless breadboard, 220, 221, 223
sort (), 36
sound
adding of to platform game, 98–99
calculating level of in spinning
cube, 129–135
getting of into computer, 177–181
recording of, 179
sox (audio Swiss Army knife), 184, 196 Space module, 106
speaking, to Pi, 180–181
special characters, 203, 205
speech recognition oracle (Pyri), 182
speech-to-text web service, 180, 196
speed_y local variable, 84
SPI (Serial Peripheral Interface) communications protocol, 231–235, 236, 239
spinbox, 59, 60
spinning cube
adding some texture, 136
addling light, 120–126
bringing it all together, 116–120
building the 3D model, 128–129
calculating sound level, 129–135
creation of, 110–116
making screen dance, 126–128
taking things further, 135–136
Sprite class, 78, 94, 99

sprites, 84–85, 86, 97
square brackets ([]), 34
square () function, 44
SS (Slave Select), 231
.ssh folder, 208
statements. See specific statements
stderr.Stdout (or standard out), 201 stdout.Stderr (or standard error), 201
stdout=subprocess.PIPE parameter, 201 storage value, 157
str (string, piece of text), 29
str () function, 32
stretching matrix, 128
string operations, 31
string type, 30
strings
converting other data types to, 32
creation of, 30
defined, 34
each level of platform game as list of, 86 stripboards, 221, 222, 223
.strip (), 216
structures, storing of values in, 33–37
subclasses, 51
subdirectories, 8
subprocess module, 158, 200–201
subprocess.call (), 184, 200, 208 subprocess.Popen (), 195, 200
sudo prefix, 11, 226
superclasses, 51, 58, 152
superuser permissions, 226
symbols_suite, 253
sys module, 101
sys.argv, 101, 167
syslog value, 157
SysStatusHandler, 157
sysstatus-template.html, 157 /sys, 199
system log file (syslog), 7, 205
System on a Chip (SoC), 109
system value, 157

T

tags, for web pages, 149
tarfile module, 214–215 tar.gz file, 212
templates, 153, 154–155, 160 terminal command, 164–165 test_ method, 252
test suites, 252–253, 255
test your knowledge

adding button to launch a

QColorDialog , 73
extending turtle controller program, 62 getting current weather, 149
of Python statements, 32–33, 38
on regular expressions, 206–207

test-drive development, 252
testing
defined, 255
getting more assertive, 250–252
how much should you test?, 254–255
purpose of, 254
regression testing, 252–253
unit tests, 245–250
test.png, 191
text, keeping of in strings, 30
text editors, Leafpad, 11, 103
text input/entry, in web browser, 66
texture, adding of to spinning cube, 136 three slashes (///), 73
3D graphics, 109, 112, 119, 136, 163,
168, 169, 173
three-dimensional float (3f) vectors, 114 time-lapse cartoon video, 192
time.sleep (), 25
timestamp, 214
Tk graphical toolkit, 57
tools, getting the best, 222–223
tornado module, 151
Tornado server, 193, 196
Tornado web application, 152, 153, 160, 194 tornado.web.RequestHandler, 152
tornado.web.StaticFileHandler, 194 touch screen, 57
training images, 190
transform_matrix variable, 114,

116, 123
transforms.compose (), 112, 128 transforms.rotation_degrees(), 113
transforms.scaling (), 113
transforms.stretching (), 113 translation matrix, 128
triangular brackets (<>), 149
True value, 31–32, 39, 42, 88, 91, 92,
226, 229, 245, 252
tuples, 33–34, 35, 36, 37, 55
Turtle class/turtle class, 49, 51 turtle module, 13–21, 49
TurtleControl class, 59, 60 turtle.Screen () statement, 49 turtle.Turtle ()
statement, 49 tweeting/tweets, 144–146
Twitter, 145, 159
2D graphics, 109, 168, 169, 173 two-dimensional list, 35
type () function, 28
type IDs, 166
type page argument, 157
typeID type, 165
types
converting between data types, 32
values as associated with, 28–29

U

uniform variables, 137
unit tests, 245–250, 252, 253, 255
unittest.main (), 247
universal character set Transformation Format

8-bit (UTF-8), 141, 246, 248
Universal Resource Locator (URL), 67
Unix systems, 214
update () method, 87, 94
upper () method, 250
URL entry bar/box/control, 67, 70, 71 urllib.request.Request (), 180 usability
problems, 255
USB devices, 205
USB keyboard, 5, 6
USB memory stick, 6, 199, 208, 229
USB microphone, 177

USB mouse, 5, 6
USB webcam, 5, 6, 184–187, 190–191, 196 USB WiFi dongle, 5, 6
use () method, 111
user-info.html, 154
user_input, 242, 244
usernames, 8, 21, 22, 33, 160
user-template.html, 154
UTF-8 (universal character set Transformation

Format 8-bit), 141, 246, 248

V

valueChanged action, 71
ValueError, 43
values

associated with types, 28–29, 55 as being numbers or text, 28
storing of in structures, 33–37

Van Loo, Gert (Raspberry Pi Foundation’s hardware supremo), 238
variable_name_getRgb () [:3], 73
variables. See also specific variables
defined, 24, 27, 55
naming convention, 28, 49
for storing information, 25
understanding, 24
/var/log folder, 205
varying keyword, 121
varying variables, 120, 129
varying vectors, 137
varying.uniform variable, 120
vector-matrix algebra, 112
vectors, 112, 120, 121, 122, 137
vertex shader, 111, 113, 115, 116,
120, 122, 137
vertex_attrib attribute, 114
vertex_glsl variable, 111, 113
vertical boxes, 63
vertices list, 111

VGA input, 6
video, making, 184–196
video encoder, 192
virtual filesystems, 199
visualiser, 130
voice control, 180
voice-driven menu, 184
voltage, 228
-v flag, 250

W

w2schools.org (HTML resource), 150 WAV files, 126, 177, 179, 184, 196 wave module, 127, 179
weather forecasts, with JSON, 147–149 Web, getting on, 149–155
web browser

adding of window menus to, 71–73 adding speech recognition oracle to, 184
creation of, 62–71

web pages, 63, 149
web server, 150
webcam viewer, 188
webcams, 6
websites, making them dynamic, 151–152 webview’s load method, 67
w:gz parameter, 215
while loops, 25, 38, 42, 55, 173, 187, 189 widgets, 59, 62, 63, 66, 71, 74
window menus, adding of, 71–73
windowing system, 7
wire cutters/strippers, 222, 223
Wireless Inventor’s Kit, 238
with block, 216
Wolfram Alpha, 181, 182
Wolfram Alpha app ID, 182
wolframalapha.com, 181
word processors, LibreOffice’s Writer, 11 World class, 87, 92
world_cube_net.png, 136 write () method, 236

X

x and y attributes, 84, 90
x coordinate (horizontal position), 112, 165 XML, 182, 188
xml module, 182

Y
y coordinate (vertical position), 112, 165
Z

z coordinate (depth), 112, 165 zoom slider bar, in web browser, 69
zoom_changed () method, 71

Science Using Python: A Computational Problem-Solving Focus

This page is intentionally left blank

Science Using Python: A Computational Problem-Solving Focus

Charles Dierbach

VP & Executive Publisher: Executive Editor:
Assistant Editor:
Marketing Manager:
Marketing Assistant:
Photo Editor:
Cover Designer:
Associate Production Manager: Production Editor:
Cover Illustration:
Don Fowley
Beth Lang Golub Samantha Mandel Christopher Ruel Ashley Tomeck Hilary Newman Thomas Nery
Joyce Poh
Jolene Ling
Norm Christiansen

This book was set in 10/12 Times LT Std by Aptara. Text and cover were printed and bound by Courier Kendallville.
This book is printed on acid free paper.

Founded in 1807, John Wiley & Sons, Inc. has been a valued source of knowledge and understanding for more than 200 years, helping people around the world meet their needs and fulfill their aspirations. Our company is built on a foundation of principles that include responsibility to the communities we serve and where we live and work. In 2008, we launched a Corporate Citizenship Initiative, a global effort to address the environmental, social, economic, and ethical challenges we face in our business. Among the issues we are addressing are carbon impact, paper specifications and procurement, ethical conduct within our business and among our vendors, and community and charitable support. For more information, please visit our website: www.wiley.com/go/citizenship.

Copyright © 2013 John Wiley & Sons, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc. 222 Rosewood Drive, Danvers, MA 01923, website www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc.,

111 River Street, Hoboken, NJ 07030-5774, (201)748-6011, fax (201)748-6008, website http://www.wiley.com/go/permissions.

Evaluation copies are provided to qualified academics and professionals for review purposes only, for use in their courses during the next academic year. These copies are licensed and may not be sold or transferred to a third party. Upon completion of the review period, please return the evaluation copy to Wiley. Return instructions and a free of charge return mailing label are available at www.wiley.com/go/returnlabel. If you have chosen to adopt this textbook for use in your course, please accept this book as your complimentary desk copy. Outside of the United States, please contact your local sales representative.

Library of Congress Cataloging-in-Publication Data
Dierbach, Charles, 1953–

Introduction to Computer Science Using Python: A Computational Problem-Solving Focus/Charles Dierbach. p. cm.
Includes index.
ISBN 978-0-470-55515-6 (pbk.)
1. Python (Computer program language) I. Title.
QA76.73.P98D547 2012
005.13'3—dc23
2012027172

Printed in the United States of America 10 9 8 7 6 5 4 3 2 1

DEDICATION

To my wife Chen Jin, and our sons Jayden and Bryson.

Brief Contents

Preface xxi Acknowledgments xxv About the Author xxvii

1 Introduction 1
2 Data and Expressions 38
3 Control Structures 79
4 Lists 125
5 Functions 168
6 Objects and Their Use 206
7 Modular Design 247

8 Text Files 289
9 Dictionaries and Sets 337
10 Object-Oriented Programming 383
11 Recursion 460
12 Computing and Its Developments 491
Appendix 525
Index 569

Contents

Preface xxi Acknowledgments xxv About the Author xxvii

1Introduction 1

MOTIVATION 2
FUNDAMENTALS 2
1.1 What Is Computer Science? 2

1.1.1 The Essence of Computational Problem Solving 3
1.1.2 Limits of Computational Problem Solving 5
Self-Test Questions 6
1.2 Computer Algorithms 6
1.2.1 What Is an Algorithm? 6
1.2.2 Algorithms and Computers: A Perfect Match 7 Self-Test Questions 8
1.3 Computer Hardware 9
1.3.1 Digital Computing: It’s All about Switches 9
1.3.2 The Binary Number System 10
1.3.3 Fundamental Hardware Components 11
1.3.4 Operating Systems—Bridging Software and Hardware 11
1.3.5 Limits of Integrated Circuits Technology: Moore’s Law 12 Self-Test Questions 13
1.4 Computer Software 14
1.4.1 What Is Computer Software? 14
1.4.2 Syntax, Semantics, and Program Translation 14
1.4.3 Procedural vs. Object-Oriented Programming 17 Self-Test Questions 17

COMPUTATIONAL PROBLEM SOLVING 17

1.5 The Process of Computational Problem Solving 17 1.5.1 Problem Analysis 18
1.5.2 Program Design 19
1.5.3 Program Implementation 21
1.5.4 Program Testing 21

1.6 The Python Programming Language 22
1.6.1 About Python 22
1.6.2 The IDLE Python Development Environment 22 1.6.3 The Python Standard Library 23
1.6.4 A Bit of Python 24
1.6.5 Learning How to Use IDLE 26

1.7 A First Program—Calculating the Drake Equation 29 1.7.1 The Problem 30
1.7.2 Problem Analysis 30
1.7.3 Program Design 30
1.7.4 Program Implementation 30
1.7.5 Program Testing 32

Chapter Summary 33
Chapter Exercises 34
Python Programming Exercises 36
Program Modification Problems 37
Program Development Problems 37

2 Data and Expressions 38

MOTIVATION 39
FUNDAMENTAL CONCEPTS 40
2.1 Literals 40

2.1.1 What Is a Literal? 40
2.1.2 Numeric Literals 40
2.1.3 String Literals 44
2.1.4 Control Characters 46
2.1.5 String Formatting 47
2.1.6 Implicit and Explicit Line Joining 48
2.1.7 Let’s Apply It—“Hello World Unicode Encoding” 48 Self-Test Questions

49
2.2 Variables and Identifiers 50
2.2.1 What Is a Variable? 50
2.2.2 Variable Assignment and Keyboard Input 52
2.2.3 What Is an Identifier? 53
2.2.4 Keywords and Other Predefined Identifiers in Python 54
2.2.5 Let's Apply It—"Restaurant Tab Calculation" 55 Self-Test Questions 56
2.3 Operators 57
2.3.1 What Is an Operator? 57
2.3.2 Arithmetic Operators 57
2.3.3 Let's Apply It—"Your Place in the Universe" 59 Self-Test Questions 60
2.4 Expressions and Data Types 61
2.4.1 What Is an Expression? 61
2.4.2 Operator Precedence 61
2.4.3 Operator Associativity 63

2.4.4 What Is a Data Type? 64
2.4.5 Mixed-Type Expressions 64
2.4.6 Let's Apply It—"Temperature Conversion Program" 65 Self-Test Questions 66

COMPUTATIONAL PROBLEM SOLVING 67
2.5 Age in Seconds Program 67
2.5.1 The Problem 67

2.5.2 Problem Analysis 67
2.5.3 Program Design 67
2.5.4 Program Implementation and Testing 69
Chapter Summary 74
Chapter Exercises 74
Python Programming Exercises 76
Program Modification Problems 76
Program Development Problems 77

3 Control Structures 79

MOTIVATION 80

FUNDAMENTAL CONCEPTS 80
3.1 What Is a Control Structure? 80
3.2 Boolean Expressions (Conditions) 81

3.2.1 Relational Operators 81
3.2.2 Membership Operators 82
3.2.3 Boolean Operators 83
3.2.4 Operator Precedence and Boolean Expressions 85
3.2.5 Short-Circuit (Lazy) Evaluation 86
3.2.6 Logically Equivalent Boolean Expressions 87 Self-Test Questions 88
3.3 Selection Control 89
3.3.1 If Statement 89
3.3.2 Indentation in Python 90
3.3.3 Multi-Way Selection 91
3.3.4 Let's Apply It—Number of Days in Month Program 94 Self-Test Questions 96
3.4 Iterative Control 96
3.4.1 While Statement 97
3.4.2 Input Error Checking 98
3.4.3 Infinite loops 99
3.4.4 Definite vs. Indefinite Loops 100
3.4.5 Boolean Flags and Indefinite Loops 100
3.4.6 Let's Apply It—Coin Change Exercise Program 101 Self-Test Questions 104

COMPUTATIONAL PROBLEM SOLVING 104
3.5 Calendar Month Program 104
3.5.1 The Problem 104

3.5.2 Problem Analysis 104
3.5.3 Program Design 105
3.5.4 Program Implementation and Testing 107 Chapter Summary 117
Chapter Exercises 118
Python Programming Exercises 120
Program Modification Problems 121
Program Development Problems 123

4 Lists 125

MOTIVATION 126
FUNDAMENTAL CONCEPTS 127
4.1 List Structures 127

4.1.1 What Is a List? 127
4.1.2 Common List Operations 127
4.1.3 List Traversal 128
Self-Test Questions 129
4.2 Lists (Sequences) in Python 130
4.2.1 Python List Type 130
4.2.2 Tuples 131
4.2.3 Sequences 132
4.2.4 Nested Lists 134
4.2.5 Let's Apply It—A Chinese Zodiac Program 135
Self-Test Questions 137
4.3 Iterating Over Lists (Sequences) in Python 137
4.3.1 For Loops 137
4.3.2 The Built-in range Function 138
4.3.3 Iterating Over List Elements vs. List Index Values 139
4.3.4 While Loops and Lists (Sequences) 140
4.3.5 Let's Apply It—Password Encryption/Decryption Program 141 Self-Test Questions 144
4.4 More on Python Lists 144
4.4.1 Assigning and Copying Lists 144
4.4.2 List Comprehensions 146

COMPUTATIONAL PROBLEM SOLVING 147
4.5 Calendar Year Program 147
4.5.1 The Problem 147

4.5.2 Problem Analysis 147
4.5.3 Program Design 148
4.5.4 Program Implementation and Testing 149
Chapter Summary 161
Chapter Exercises 162
Python Programming Exercises 164
Program Modification Problems 164

5Functions 168

MOTIVATION 169
FUNDAMENTAL CONCEPTS 169
5.1 Program Routines 169

5.1.1 What Is a Function Routine? 169
5.1.2 Defining Functions 170
5.1.3 Let’s Apply It—Temperature Conversion Program (Function Version) 173
Self-Test Questions 175
5.2 More on Functions 176
5.2.1 Calling Value-Returning Functions 176
5.2.2 Calling Non-Value-Returning Functions 177
5.2.3 Parameter Passing 178
5.2.4 Keyword Arguments in Python 181
5.2.5 Default Arguments in Python 183
5.2.6 Variable Scope 183
5.2.7 Let’s Apply It—GPA Calculation Program 186
Self-Test Questions 189

COMPUTATIONAL PROBLEM SOLVING 189
5.3 Credit Card Calculation Program 189
5.3.1 The Problem 189
5.3.2 Problem Analysis 190
5.3.3 Program Design 190
5.3.4 Program Implementation and Testing 191
Chapter Summary 202
Chapter Exercises 202
Python Programming Exercises 203
Program Modification Problems 204
Program Development Problems 204

6 Objects and Their Use 206

MOTIVATION 207
FUNDAMENTAL CONCEPTS 207
6.1 Software Objects 207

6.1.1 What Is an Object? 208

6.1.2 Object References 209
Self-Test Questions 216
6.2 Turtle Graphics 216
6.2.1 Creating a Turtle Graphics Window 216
6.2.2 The “Default” Turtle 218
6.2.3 Fundamental Turtle Attributes and Behavior 219
6.2.4 Additional Turtle Attributes 222
6.2.5 Creating Multiple Turtles 225
6.2.6 Let’s Apply It—Bouncing Balls Program 226 Self-Test Questions 229

COMPUTATIONAL PROBLEM SOLVING 229 6.3 Horse Race Simulation Program 229
6.3.1 The Problem 230
6.3.2 Problem Analysis 230
6.3.3 Program Design 231
6.3.4 Program Implementation and Testing 231
Chapter Summary 243
Chapter Exercises 243
Python Programming Exercises 244
Program Modification Problems 245
Program Development Problems 246

7 Modular Design 247

MOTIVATION 248
FUNDAMENTAL CONCEPTS 248
7.1 Modules 248

7.1.1 What Is a Module? 248
7.1.2 Module Specification 249
Self-Test Questions 251
7.2 Top-Down Design 251
7.2.1 Developing a Modular Design of the Calendar Year Program 251
7.2.2 Specification of the Calendar Year Program Modules 252 Self-Test Questions 255
7.3 Python Modules 255
7.3.1 What Is a Python Module? 255
7.3.2 Modules and Namespaces 256
7.3.3 Importing Modules 257

7.3.4 Module Loading and Execution 260
7.3.5 Local, Global, and Built-in Namespaces in Python 262
7.3.6 A Programmer-Defined Stack Module 264
7.3.7 Let's Apply It—A Palindrome Checker Program 267 Self-Test Questions 268

COMPUTATIONAL PROBLEM SOLVING 269
7.4 Calendar Year Program (function version) 269
7.4.1 The Problem 269
7.4.2 Problem Analysis 269
7.4.3 Program Design 269
7.4.4 Program Implementation and Testing 269
Chapter Summary 284
Chapter Exercises 284
Python Programming Exercises 286
Program Modification Problems 287

8 Text Files 289

MOTIVATION 290
FUNDAMENTAL CONCEPTS 290
8.1 What Is a Text File? 290
8.2 Using Text Files 291

8.2.1 Opening Text Files 291
8.2.2 Reading Text Files 293
8.2.3 Writing Text Files 294
Self-Test Questions 295
8.3 String Processing 296
8.3.1 String Traversal 296
8.3.2 String-Applicable Sequence Operations 296
8.3.3 String Methods 297
8.3.4 Let's Apply It—Sparse Text Program 300
Self-Test Questions 303
8.4 Exception Handling 303
8.4.1 What Is an Exception? 303
8.4.2 The Propagation of Raised Exceptions 304
8.4.3 Catching and Handling Exceptions 305

8.4.4 Exception Handling and User Input 307
8.4.5 Exception Handling and File Processing 309
8.4.6 Let’s Apply It—Word Frequency Count Program 310 Self-Test Questions 314

COMPUTATIONAL PROBLEM SOLVING 314
8.5 Cigarette Use/Lung Cancer Correlation Program 314
8.5.1 The Problem 315
8.5.2 Problem Analysis 315
8.5.3 Program Design 316
8.5.4 Program Implementation and Testing 318
8.5.5 Determining the Correlation Between Smoking and Lung Cancer 331
Chapter Summary 331
Chapter Exercises 332
Python Programming Exercises 333
Program Modification Problems 333
Program Development Problems 334

9 Dictionaries and Sets 337

MOTIVATION 338
FUNDAMENTAL CONCEPTS 338
9.1 Dictionary Type in Python 338

9.1.1 What Is a Dictionary? 339
9.1.2 Let’s Apply It—Phone Number Spelling Program 342 Self-Test Questions 346
9.2 Set Data Type 346
9.2.1 The Set Data Type in Python 346
9.2.2 Let’s Apply It—Kitchen Tile Visualization Program 348 Self-Test Questions 356

COMPUTATIONAL PROBLEM SOLVING 356
9.3 A Food Co-op’s Worker Scheduling Simulation 356
9.3.1 The Problem 357
9.3.2 Problem Analysis 357
9.3.3 Program Design 358
9.3.4 Program Implementation and Testing 360
9.3.5 Analyzing a Scheduled vs. Unscheduled Co-op Worker Approach 375

Chapter Summary 379
Chapter Exercises 379
Python Programming Exercises 380
Program Modification Problems 380
Program Development Problems 381

10 Object-Oriented Programming 383

MOTIVATION 384
FUNDAMENTAL CONCEPTS 384
10.1 What Is Object-Oriented Programming? 384

10.1.1 What Is a Class? 385
10.1.2 Three Fundamental Features of Object-Oriented Programming 385

10.2 Encapsulation 386
10.2.1 What Is Encapsulation? 386
10.2.2 Defining Classes in Python 387
10.2.3 Let's Apply It—A Recipe Conversion Program 394 Self-Test Questions 399

10.3 Inheritance 400
10.3.1 What Is Inheritance? 400
10.3.2 Subtypes 401
10.3.3 Defining Subclasses in Python 402
10.3.4 Let's Apply It—A Mixed Fraction Class 407
Self-Test Questions 411

10.4 Polymorphism 411
10.4.1 What Is Polymorphism? 411
10.4.2 The Use of Polymorphism 414
Self-Test Questions 417

10.5 Object-Oriented Design Using UML 417
10.5.1 What Is UML? 417
10.5.2 UML Class Diagrams 418
Self-Test Questions 422

COMPUTATIONAL PROBLEM SOLVING 423
10.6 Vehicle Rental Agency Program 423

10.6.1 The Problem 423
Contents xvii

10.6.2 Problem Analysis 423
10.6.3 Program Design 423
10.6.4 Program Implementation and Testing 429 Chapter Summary 453
Chapter Exercises 454
Python Programming Exercises 455
Program Modification Problems 456
Program Development Problems 457

11 Recursion 460

MOTIVATION 461
FUNDAMENTAL CONCEPTS 461
11.1 Recursive Functions 461

11.1.1 What Is a Recursive Function? 461
11.1.2 The Factorial Function 464
11.1.3 Let's Apply It—Fractals (Sierpinski Triangle) 467 Self-Test Questions 471
11.2 Recursive Problem Solving 472
11.2.1 Thinking Recursively 472
11.2.2 MergeSort Recursive Algorithm 472
11.2.3 Let's Apply It—MergeSort Implementation 474 Self-Test Questions 476
11.3 Iteration vs. Recursion 476

COMPUTATIONAL PROBLEM SOLVING 477
11.4 Towers of Hanoi 477
11.4.1 The Problem 477
11.4.2 Problem Analysis 477
11.4.3 Program Design and Implementation 481
Chapter Summary 487
Chapter Exercises 487
Python Programming Exercises 488
Program Modification Problems 489
Program Development Problems 490

12 Computing and Its Developments 491

CONTRIBUTIONS TO THE MODERN COMPUTER 492
12.1 The Concept of a Programmable Computer 492
12.1.1 “Father of the Modern Computer”—Charles Babbage (1800s) 492 12.1.2 “The First Computer Programmer”—Ada Lovelace (1800s) 493
12.2 Developments Leading to Electronic Computing 493
12.2.1 The Development of Boolean Algebra (mid-1800s) 493 12.2.2 The Development of the Vacuum Tube (1883) 494 12.2.3 The Development of Digital Electronic Logic Gates (1903) 494

12.2.4 The Development of Memory Electronic Circuits (1919) 495
12.2.5 The Development of Electronic Digital Logic Circuits (1937) 495
12.2.6 “The Father of Information Theory”—Claude Shannon (1948) 496

FIRST-GENERATION COMPUTERS (1940s–mid-1950s) 496
12.3 The Early Groundbreakers 496
12.3.1 The Z3—The First Programmable Computer (1941) 496
12.3.2 The Mark I—First Computer Project in the United States (1937–1943) 497 12.3.3 The ABC—The First Fully Electronic Computing Device (1942) 498
12.3.4 Colossus—A Special-Purpose Electronic Computer (1943) 499 12.3.5 ENIAC—The First Fully Electronic Programmable Computer 500 12.3.6 EDVAC/ACE—The First Stored Program Computers (1950) 501 12.3.7 Whirlwind—The First Real-Time Computer (1951) 502
12.4 The First Commercially Available Computers 503
12.4.1 The Struggles of the Eckert-Mauchly Computer Corporation (1950) 503
12.4.2 The LEO Computer of the J. Lyons and Company (1951) 504

SECOND-GENERATION COMPUTERS (mid-1950s to mid-1960s) 505 12.5 Transistorized Computers 505
12.5.1 The Development of the Transistor (1947) 505
12.5.2 The First Transistor Computer (1953) 506
12.6 The Development of High-Level Programming Languages 506
12.6.1 The Development of Assembly Language (early 1950s) 506 12.6.2 The First High-Level Programming Languages (mid-1950s) 507 12.6.3 The First “Program Bug” (1947) 508

THIRD-GENERATION COMPUTERS (mid-1960s to early 1970s) 508 12.7 The Development of the Integrated Circuit (1958) 508

12.7.1 The Catalyst for Integrated Circuit Advancements (1960s) 509 12.7.2 The Development of the Microprocessor (1971) 511
12.8 Mainframes, Minicomputers, and Supercomputers 512
12.8.1 The Establishment of the Mainframe Computer (1962) 512 12.8.2 The Development of the Minicomputer (1963) 513
12.8.3 The Development of the UNIX Operating System (1969) 513 12.8.4 The Development of Graphical User Interfaces (early 1960s) 514 12.8.5 The Development of the Supercomputer (1972) 515

FOURTH-GENERATION COMPUTERS (early 1970s to the Present) 515 12.9 The Rise of the Microprocessor 515
12.9.1 The First Commercially Available Microprocessor (1971) 515 12.9.2 The First Commercially Available Microcomputer Kit (1975) 516
12.10 The Dawn of Personal Computing 516
12.10.1 The Beginnings of Microsoft (1975) 516
12.10.2 The Apple II (1977) 517
12.10.3 IBM's Entry into the Microcomputer Market (1981) 517 12.10.4 Society Embraces the Personal Computer (1983) 518
12.10.5 The Development of Graphical User Interfaces (GUIs) 518 12.10.6 The Development of the C11 Programming Language 519

THE DEVELOPMENT OF COMPUTER NETWORKS 520
12.11 The Development of Wide Area Networks 520
12.11.1 The Idea of Packet-Switched Networks (early 1960s) 520 12.11.2 The First Packet-Switched Network: ARPANET (1969) 520
12.12 The Development of Local Area Networks (LANs) 521
12.12.1 The Need for Local Area Networks 521
12.12.2 The Development of Ethernet (1980) 521
12.13 The Development of the Internet and World Wide Web 522
12.13.1 The Realization of the Need for "Internetworking" 522 12.13.2 The Development of the TCP/IP Internetworking Protocol (1973) 522 12.13.3 The Development of the World Wide Web (1990) 522 12.13.4 The Development of the Java Programming Language (1995) 523

Appendix 525 Index 569
This page is intentionally left blank

Preface

Book Concept

This text introduces students to programming and computational problem solving using the Python 3 programming language. It is intended primarily for a first-semester computer science (CS1) course, but is also appropriate for use in any course providing an introduction to computer programming and/or computational problem solving. The book provides a step-by-step, “hands on” pedagogical approach which, together with Python’s clear and simple syntax, makes this book easy to teach and learn from.

The primary goal in the development of this text was to create a pedagogically sound and accessible textbook that emphasizes fundamental programming and computational problem- solving concepts over the minutiae of a particular programming language. Python’s ease in the creation and use of both indexed and associative data structures (in the form of lists/tuples and dictionaries), as well as sets, allows for programming concepts to be demonstrated without the need for detailed discussion of programming language specifics.

Taking advantage of Python’s support of both the imperative (i.e., procedural) and objectoriented paradigms, a “back to basics,” “objects-late” approach is taken to computer programming. It follows the belief that solid grounding in imperative programming should precede the larger number of (and more abstract) concepts of the object-oriented paradigm. Therefore, objects are not covered until Chapter 5, and object-oriented programming is not introduced until Chapter 10. For those who do not wish to introduce object-oriented programming, Chapter 10 can easily be skipped.

How This Book Is Different
This text has a number of unique pedagogical features including:

♦ A *short motivation section* at the beginning of each chapter which provides a larger perspective on the chapter material to be covered.
♦ *Hands-on exercises* throughout each chapter which take advantage of the interactive capabilities of Python.
♦ A fully-developed computational problem solving example at the end of each chapter that places an emphasis on *program testing and program debugging*.

♦ A richly illustrated, final chapter on "*Computing and Its Developments*" that provides a storyline of notable individuals, accomplishments, and developments in computing, from Charles Babbage through modern times.

♦ A *Python 3 Programmers' Reference* in the back of the text which allows the book to serve as both a pedagogical resource *and* as a convenient Python reference.

Pedagogical Features

The book takes a step-by-step pedagogical approach. Each new concept is immediately followed by a pedagogical element that elucidates the material covered and/or challenges students' understanding. The specific pedagogical features of the book are listed below.

Summary Boxes

At the end of each subsection, a clearly outlined summary box is provided containing the most salient information of the material just presented. These concise summaries also serve as useful reference points as students review the chapter material.

Let's Try It Sections

Also at the end of each subsection, a short Let's Try It section is given in which students are asked to type in Python code (into the Python shell) and observe the results. These "hands on" exercises help to immediately reinforce material as students progress through the chapter.

Let's Apply It Sections

At the end of each major section, a complete program example is provided with detailed line-by-line discussion. These examples serve to demonstrate the programming concepts just learned in the context of an actual program.

Self-Test Questions

Also at the end of each major section, a set of multiple-choice/short-answer questions is given. The answers are included so that students may perform a comprehension self check in answering these.

A variety of exercises and program assignments are also included at the end of every chapter. These are designed to gradually ease students from review of general concepts, to code-writing exercises, to modification of significant-sized

programs, to developing their own programs, as outlined below.

Chapter Exercises
At the end of each chapter, a set of simple, short-answer questions are provided.
Python Programming Exercises
Also at the end of each chapter, a set of simple, short Python programming exercises are given.

Program Modification Problems
Additionally, at the end of each chapter is a set of programming problems in which students are asked to make various modifications to program examples in the chapter. Thus, these exercises do not require students to develop a program from scratch. They also serve as a means to encourage students to think through the chapter program examples.

Program Development Problems
Finally, at the end of each chapter is a set of computational problems which students are to develop programs for from scratch. These problems are generally similar to the program examples given in the chapters.

Emphasis on Computational Problem Solving

The capstone programs at the end of each chapter show students how to go through the process of computational problem solving. This includes problem analysis, design, implementation, and testing, as outlined in Chapter 1. As a program is developed and tested, errors are intentionally placed in the code, leading to discussion and demonstration of program testing and debugging. Programming errors, therefore, are presented as a normal part of software development. This helps students develop

their own program debugging skills, and reinforces the idea that program debugging is an inevitable part of program development—an area of coverage that is crucial for beginning programmers, and yet often lacking in introductory computer science books.

Given the rigor with which these problems are presented, the sections are somewhat lengthy. Since the capstones do not introduce any new concepts, they may be skipped if the instructor does not have time to cover them.

Guided Book Tour

Chapter 1 addresses the question “What is computer science?” Computational problem solving is introduced, including discussions on the limits of computation, computer algorithms, computer hardware, computer software, and a brief introduction to programming in Python. The end of the chapter lays out a step-by-step computational problem solving process for students consisting of problem analysis, program design, program implementation, and program testing.

Chapter 2 covers data and expressions, including arithmetic operators, discussion of limits of precision, output formatting, character encoding schemes, control characters, keyboard input, operator precedence and associativity, data types, and coercion vs. type conversion.

Chapter 3 introduces control structures, including relational, membership and Boolean operators, short-circuit (lazy) evaluation, selection control (if statements) and indentation in Python, iterative control (while statements), and input error checking. (For statements are not covered until Chapter 4 on Lists.) Break and continue statements are not introduced in this book. It is felt that these statements, which violate the principles of structured programming, are best not introduced to the beginning programmer.

Chapter 4 presents lists and for statements. The chapter opens with a general discussion of lists and list operations. This is followed by lists and tuples in Python, including nested lists, for loops, the built-in range function, and list comprehensions. Since all values in Python are (object) references, and lists are the first mutable type to which students are introduced, a discussion of shallow vs. deep copying is provided without explicit mention of references. The details of object representation in Python is covered in Chapter 6.

Chapter 5 introduces the notion of a program routine, including discussions of parameter passing (actual argument vs. formal parameters), value vs. non-value returning functions, mutable vs. immutable arguments, keyword and default arguments in Python, and local vs. global scope.

Chapter 6 introduces students to the concept of objects in programming. Here students see how objects are represented as references, and therefore are able to fully understand the behavior of assignment and copying of lists (initially

introduced in Chapter 4), as well as other types. Turtle graphics is introduced (by use of the turtle module of the Python Standard Library) and is used to provide an intuitive, visual means of understanding the concept of object instances. This also allows students to write fun, graphical programs, while at the same time reinforcing the notion and behavior of objects.

Chapter 7 covers modules and modular design. It starts off by explaining the general notion of a module and module specification, including docstrings in Python. It is followed by a discussion of topdown design. It then introduces modules in Python, including namespaces, the importing of modules, module private variables, module loading and execution, and local, global, and built-in namespaces. The notion of a stack is introduced here via the development of a programmer-defined stack module.

Chapter 8 introduces text files and string processing. It starts with how to open/close, and read/ write files in Python. Then the string-applicable sequence operations from Chapter 4 are revisited, and additional string methods are covered. Exception handling is introduced in the context of file handling, and some of the more commonly occurring Python Standard Exceptions are introduced.

Chapter 9 presents dictionaries (Python's associative data structure) and sets.

Chapter 10 introduces object-oriented programming. It begins with a discussion of classes, and the notion of encapsulation. Then, how classes are defined is presented, including discussion of special methods in Python. Inheritance and subtypes are discussed next, followed by a discussion of the use of polymorphism. Finally, the chapter ends with a brief introduction to class diagrams in UML.

Chapter 11 covers recursion and recursive problem solving, including discussion of recursion vs. iteration, and when recursion is appropriately used.

Chapter 12 concludes the book by providing an overview of the people, achievements and developments in computing. This chapter serves to "humanize" the field and educate students on the history of the discipline.

Online Textbook Supplements

All supplements are available via the book's companion website at www.wiley.com/college/dierbach. Below is the list of supplements that accompany this text:

♦ Instructor's manual, with answers to all exercises and program assignments
♦ PowerPoint slides, summarizing the key points of each chapter
♦ Program code for all programs in the book
♦ Test bank of exam questions for each chapter

A separate student companion site is available at the above web site which grants students access to the program code and additional files needed to execute and/or modify programs in the book. All other program code is available to instructors only.

Acknowledgments

I would first like to thank the people at Wiley & Sons. To Dan Sayre, for getting this project going; to my editor Beth Golub, for all her patience and guidance during the evolution of the book; and to Samantha Mandel, Assistant Editor, for her invaluable help. I would also like to thank Harry Nolan, Design Director, who took the time to ensure that the book design turned out as I envisioned; to Jolene Ling, Production Editor, who so graciously worked on the production of the book and the ensuing changes, and for seeing that everything came together.

There are many others to thank who have in some way contributed to this project. First, thanks to Harry Hochheiser, for all the motivating and informative discussions that eventually led me to Python, and ultimately the development of this book. Many thanks to my colleague Josh Dehlinger, who lent his extremely critical eye to the project (and took up the slack on many of my department duties!). Thanks to my department chair, Chao Lu, for his support, friendship, and for creating such a collegial and productive environment to work (and for funneling some of my duties to Josh!). And thanks to Shiva Azadegan, who first planted the idea of writing a book in my head, and for being such a supportive friend, as well as a wonderful colleague.

I would also like to acknowledge a couple of my outstanding graduate TAs in the Python course for all their help and enthusiasm on the project. I thank Crystal McKinney, for so freely offering her time to review chapters and offer her suggestions. I owe a great debt of thanks to Leela Sedaghat, who contributed to

the project in so many ways—her insightful review of chapters, the enormous amount of time spent on verifying and obtaining image permissions, and her design of and contribution to the Python Programmers' Reference manual, which without her help, would never have been completed on time. Previous graduate students Ahbi Grover and Lanlan Wang also read earlier drafts of the book.

Finally, I thank the reviewers. Without them, the book could never be what it is now. First, special thanks to Claude Anderson of Rose-Hulman Institute of Technology. His meticulous review for technical errors, and his suggestions on pedagogy, have significantly contributed to the book. In addition, I thank each of the following individuals who served as reviewers on this project: James Atlas, University of Delaware; Richard Borie, University of Alabama; Tim Bower, Kansas State University Salina; Darin Brezeale, University of Texas at Arlington; Diana Cukierman, Simon Fraser University; Chris Heiden, St. Clair County Community College; Ric Heishman, George Mason University; Jennifer Kay, Rowan University; Debby Keen, University of Kentucky; Clayton Lewis, University of Colorado; Alan McLeod, Queen's University at Kingston; Ethan Miller, University of California, Santa Cruz; Joe Oldham, Centre College; Susan Mary Rosselet, Bemidji State University; Terry A. Scott, University of Northern Colorado; and Leon Tietz, Minnesota State University Mankato.

About the Author

Charles Dierbach is an Associate Professor of computer science at Towson University, and has regularly taught introductory undergraduate computer science courses for the past thirty-five years. He received his Ph.D. in Computer Science from the University of Delaware. While a lecturer there, he received the Outstanding Teaching Award from the undergraduate chapter of the ACM. At Towson, he served as Director of the Undergraduate Computer Science program for over ten years. In addition to teaching introductory computer science courses, Dr. Dierbach also teaches undergraduate and graduate courses in object-oriented design and programming.

Introduction CHAPTER 1

This chapter addresses the question "What is computer science?" We begin by introducing the essence of computational problem solving via some classic examples. Next, computer algorithms, the heart of computational problem solving, are discussed. This is followed by a look at computer hardware (and the related issues of binary representation and operating systems) and computer software (and the related issues of syntax, semantics, and program translation). The chapter finishes by presenting the process of computational problem solving, with an introduction to the Python programming language.

OBJECTIVES

After reading this chapter and completing the exercises, you will be able to: ♦ Explain the essence of computational problem solving ♦ Explain what a computer algorithm is

♦ Explain the fundamental components of digital hardware ♦ Explain the role of binary representation in digital computing ♦ Explain what an operating systems is

♦ Explain the fundamental concepts of computer software ♦ Explain the fundamental features of IDLE in Python

♦ Modify and execute a simple Python program

CHAPTER CONTENTS

Motivation
Fundamentals
1.1 What Is Computer Science?
1.2 Computer Algorithms
1.3 Computer Hardware
1.4 Computer Software
Computational Problem Solving
1.5 The Process of Computational Problem Solving 1.6 The Python Programming Language

1.7 A First Program—Calculating the Drake Equation 1

MOTIVATION

Computing technology has changed, and is continuing to change the world. Essentially every aspect of life has been impacted by computing. Just-in-time inventory allows companies to significantly reduce costs. Universal digital medical records promise to save the lives of many of the estimated 100,000 people who die each year from medical errors. Vast information resources, such

as Wikipedia, now provide easy, quick access to a breadth of knowledge as never before. Information sharing via Facebook and Twitter has not only brought family and friends together in new ways, but has also helped spur political change around the world. New interdisciplinary fields combining computing and science will lead to breakthroughs previously unimaginable. Computing-related fields in almost all areas of study are emerging (see Figure 1-1).

In the study of computer science, there are fundamental principles of computation to be learned that will never change. In addition to these principles, of course, there is always changing technology. That is what makes the field of computer science so exciting. There is constant change and advancement, but also a foundation of principles to draw from. What can be done with computation is limited only by our imagination. With that said, we begin our journey into the world of computing. I have found it an unending fascination—I hope that you do too. Bon voyage!

Various Computational-Related Fields		
Computational Biology	Computational Medicine	Computational Journalism
Computational Chemistry	Computational Pharmacology	Digital Humanities
Computational Physics	Computational Economics	Computational Creativity
Computational Mathematics	Computational Textiles	Computational Music
Computational Materials Science	Computational Architecture	Computational Photography
Computer-Aided Design	Computational Social Science	Computational Advertising
Computer-Aided Manufacturing	Computational Psychology	Computational Intelligence

FIGURE 1-1 Computing-Related Specialized Fields

FUNDAMENTALS

1.1 What Is Computer Science?

Many people, if asked to define the field of computer science, would likely say that it is about programming computers. Although programming is certainly a primary activity of computer science, programming languages and computers are only *tools*. What computer science is fundamentally about is **computational problem solving**—that is, solving problems by the use of computation (Figure 1-2).

This description of computer science provides a succinct definition of the field. However, it does not convey its tremendous *breadth and diversity*. There are various areas of study in computer science including software engineering (the design and implementation of large software systems), database management, computer networks, computer graphics, computer simulation, data mining, information security, programming language design, systems programming, computer architecture, human–computer interaction, robotics, and artificial intelligence, among others.

The definition of computer science as computational problem solving begs the question: *What is computation?* One characterization of computation is given by the notion of an *algorithm*. The definition of an algorithm is given in section 1.2. For now, consider an algorithm to be a series of steps that can be systematically followed for producing the answer to a certain type of problem. We look at fundamental issues of computational problem solving next.

FIGURE 1-2 Computational Problem Solving

Computer science is fundamentally about computational problem solving.

1.1.1 The Essence of Computational Problem Solving

In order to solve a problem computationally, two things are needed: a *representation* that captures all the relevant aspects of the problem, and an *algorithm* that solves the problem by use of the representation. Let's consider a problem known as the **Man, Cabbage, Goat, Wolf problem**(Figure 1-3).

A man lives on the east side of a river. He wishes to bring a cabbage, a goat, and a wolf to a village on the west side of the river to sell. However, his boat is only big enough to hold himself, and either the cabbage, goat, or wolf. In addition, the man cannot leave the goat alone with the cabbage because the goat will eat the cabbage, and he cannot leave the wolf alone with the goat because the wolf will eat the goat. How does the man solve his problem?

There is a simple algorithmic approach for solving this problem by simply trying all possible combinations of items that may be rowed back and forth across the river. Trying all possible solutions to a given problem is referred to as a *brute*

force approach. What would be an appropriate

FIGURE 1-3 Man, Cabbage, Goat, Wolf Problem

representation for this problem? Since only the relevant aspects of the problem need to be represented, all the irrelevant details can be omitted. A representation that leaves out details of what is being represented is a form of **abstraction**.

The use of abstraction is prevalent in computer science. In this case, is the color of the boat relevant? The width of the river? The name of the man? No, the only relevant information is *where* each item is at each step. The collective location of each item, in this case, refers to the *state* of the problem. Thus, the *start state* of the problem can be represented as follows.

man cabbage goat wolf [E, E, E, E]

In this representation, the symbol E denotes that each corresponding object is on the east side of the river. If the man were to row the goat across with him, for

example, then the representation of the new problem state would be

man cabbage goat wolf [W, E, W, E]

in which the symbol W indicates that the corresponding object is on the west side of the river—in this case, the man and goat. (The locations of the cabbage and wolf are left unchanged.) A solution to this problem is a sequence of steps that converts the initial state,

[E, E, E, E]
in which all objects are on the east side of the river, to the *goal state*,
[W, W, W, W]

in which all objects are on the west side of the river. Each step corresponds to the man rowing a particular object across the river (or the man rowing alone). As you will see, the Python programming language provides an easy means of representing sequences of values. The remaining task is to develop or find an existing algorithm for computationally solving the problem using this representation. The solution to this problem is left as a chapter exercise.

As another example computational problem, suppose that you needed to write a program that displays a calendar month for any given month and year, as shown in Figure 1-4. The representation of this problem is rather straightforward. Only a few values need to be maintained—the month and year, the number of days in each month, the names of the days of the week, and the day of the week that the first day of the month falls on. Most of these values are either provided by the user (such as the month and year) or easily determined (such as the number of days in a given month).

The less obvious part of this problem is how to determine the day of the week that a given date falls on. You would need an algorithm that can compute this. Thus, no matter how well you may know a given programming language or how good a programmer you may be, without such an algo

rithm you could not solve this problem.

MAY 2012

Sun	Mon	Tues	Wed	Thur	Fri	Sat
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

FIGURE 1-4 Calendar Month

In order to solve a problem computationally, two things are needed: a *representation* that captures all the relevant aspects of the problem, and an *algorithm* that solves the problem by use of the representation.

1.1.2 Limits of Computational Problem Solving

Once an algorithm for solving a given problem is developed or found, an important question is, “Can a solution to the problem be found in a reasonable amount of time?” If not, then the particular algorithm is of limited practical use.

San Francisco Boston 2708 mi.
New York Chicago
344 mi.

748 mi. 1749 mi.
Los Angeles$_{\textbf{Atlanta}}$
FIGURE 1-5 Traveling Salesman Problem

The **Traveling Salesman problem** (Figure 1-5) is a classic computational

problem in computer science. The problem is to find the shortest route of travel for a salesman needing to visit a given set of cities. In a brute force approach, the lengths of all possible routes would be calculated and compared to find the shortest one. For ten cities, the number of possible routes is 10! (10 factorial), or over three and a half million (3,628,800). For twenty cities, the number of possible routes is 20!, or over two and a half quintillion (2,432,902,008,176,640,000). If we assume that a computer could compute the lengths of one million routes per second, it would take over 77,000 years to find the shortest route for twenty cities by this approach. For 50 cities, the number of possible routes is over 10^{64}. In this case, it would take more time to solve than the age of the universe!

A similar problem exists for the game of chess (Figure 1-6). A brute force approach for a chess-playing program would be to "look ahead" to *all* the eventual outcomes of every move that

can be made in deciding each next move. There

FIGURE 1-6 Game of Chess

are approximately 10^{120} possible chess games that can be played. This is related to the average number of look-ahead steps needed for deciding each move. How big is this number? There are approximately 10^{80} atoms in the observable universe, and an estimated 3 3 10^{90} grains of sand to fill the universe solid.

Thus, *there are more possible chess games that can be played than grains of sand to fill the universe solid!* For problems such as this and the Traveling Salesman problem in which a brute-force approach is impractical to use, more efficient problem-solving methods must be discovered that find either an exact or an approximate solution to the problem.

Any algorithm that correctly solves a given problem must solve the problem in a reasonable amount of time, otherwise it is of limited practical use.

Self-Test Questions

1. A good definition of computer science is "the science of programming computers." (TRUE/FALSE)

2. Which of the following areas of study are included within the field of computer science? **(a)** Software engineering
(b) Database management
(c) Information security
(d) All of the above

3. In order to computationally solve a problem, two things are needed: a representation of the problem, and an _____ that solves it.
4. Leaving out detail in a given representation is a form of _____.

5. A "brute-force" approach for solving a given problem is to:
(a) Try all possible algorithms for solving the problem.
(b) Try all possible solutions for solving the problem.
(c) Try various representations of the problem.
(d) All of the above

6. For which of the following problems is a brute-force approach practical to use? **(a)** Man, Cabbage, Goat, Wolf problem
(b) Traveling Salesman problem
(c) Chess-playing program
(d) All of the above

ANSWERS: 1. False, 2. (d), 3. algorithm, 4. abstraction, 5. (b), 6. (a)

1.2 Computer Algorithms

This section provides a more complete description of an algorithm than given above, as well as an example algorithm for determining the day of the week for a given date.

1.2.1 What Is an Algorithm?

An **algorithm** is a finite number of clearly described, unambiguous “doable” steps that can be systematically followed to produce a desired result for given input in a finite amount of time (that is, it

eventually terminates). Algorithms solve *general* problems (determining whether any given number is a prime number), and not specific ones (determining whether 30753 is a prime number). Algorithms, therefore, are general computational methods used for solving particular problem instances.

The word “algorithm” is derived from the ninth-century Arab mathematician, Al-Khwarizmi (Figure 1-7), who worked on “written processes to achieve some goal.” (The term “algebra” also derives from the term “al-jabr,” which he introduced.)

Computer algorithms are central to computer science. They provide step-by-step methods of computation that a machine can carry out. Having high-speed machines (computers) that can consistently follow and execute a given set of instructions provides a reliable and effective means of realizing computation. However, *the computation that a given computer performs is only as good as the underlying algorithm used*. Understanding what can be effectively programmed and executed by computers, therefore, relies on the understanding of computer algorithms.

FIGURE 1-7 Al-Khwarizmi (Ninth Century A.D.)

An **algorithm** is a finite number of clearly described, unambiguous “doable” steps that can be systematically followed to produce a desired result for given input in a finite amount of time.

1.2.2 Algorithms and Computers: A Perfect Match

Much of what has been learned about algorithms and computation since the beginnings of modern computing in the 1930s–1940s could have been studied centuries ago, since the study of algorithms does not depend on the existence of computers. The algorithm for performing long division is such an example. However, most algorithms are not as simple or practical to apply manually. Most require the use of computers either because they would require too much time for a person to apply, or involve so much detail as to make human error likely. Because *computers can execute instructions very quickly and reliably without error*, algorithms and computers are a perfect match! Figure 1-8 gives an example algorithm for determining the day of the week for any date between

January 1, 1800 and December 31, 2099.

Because computers can execute instructions very quickly and reliably without error, algorithms and computers are a perfect match.

To determine the day of the week for a given **month**, **day**, and **year**:

1. Let **century_digits** be equal to the first two digits of the year.
2. Let **year_digits** be equal to the last two digits of the year.
3. Let **value** be equal to **year_digits +** floor(**year_digits** / 4)
4. If **century_digits** equals 18, then add 2 to **value**, else
   if **century_digits** equals 20, then add 6 to **value**.
5. If the **month** is equal to January and **year** is not a leap year,
   then add 1 to **value**, else,

   if the **month** is equal to February and the **year** is a leap year, then
   add 3 to **value**; if not a leap year, then add 4 to **value,** else,

   if the **month** is equal to March or November, then add 4 to **value**, else,

   if the **month** is equal to May, then add 2 to **value**, else,

   if the **month** is equal to June, then add 5 to **value**, else,

   if the **month** is equal to August, then add 3 to **value**, else,

   if the **month** is equal to October, then add 1 to **value**, else,

   if the **month** is equal to September or December, then add 6 to **value**,
6. Set **value** equal to (**value** + **day**) mod 7.
7. If **value** is equal to 1, then the day of the week is Sunday; else

   if **value** is equal to 2, day of the week is Monday; else

   if **value** is equal to 3, day of the week is Tuesday; else

   if **value** is equal to 4, day of the week is Wednesday; else

   if **value** is equal to 5, day of the week is Thursday; else

   if **value** is equal to 6, day of the week is Friday; else

   if **value** is equal to 0, day of the week is Saturday

FIGURE 1-8 Day of the Week Algorithm
Note that there is no value to add for the months of April and July.
Self-Test Questions

1. Which of the following are true of an algorithm?
(a) Has a finite number of steps
(b) Produces a result in a finite amount of time

(c) Solves a general problem
(d) All of the above

2. Algorithms were first developed in the 1930–1940s when the first computing machines appeared. (TRUE/FALSE)

3. Algorithms and computers are a “perfect match” because: (Select all that apply.) **(a)** Computers can execute a large number of instructions very quickly.
(b) Computers can execute instructions reliably without error.
(c) Computers can determine which algorithms are the best to use for a given problem.

4. Given that the year 2016 is a leap year, what day of the week does April 15 th of that year fall on? Use the algorithm in Figure 1-8 for this.

5. Which of the following is an example of an algorithm? (Select all that apply.)
(a) A means of sorting any list of numbers
(b) Directions for getting from your home to a friend’s house
(c) A means of finding the shortest route from your house to a friend’s house.

ANSWERS: 1. (d), 2. False, 3. (a,b) 4. Friday, 5. (a,c)

1.3 Computer Hardware

Computer hardware comprises the physical part of a computer system. It includes the all-important components of the *central processing unit* (CPU) and *main memory*. It also includes *peripheral components* such as a keyboard, monitor, mouse, and printer. In this section, computer hardware and the intrinsic use of binary representation in computers is discussed.

1.3.1 Digital Computing: It’s All about Switches

It is essential that computer hardware be reliable and error free. If the hardware gives incorrect results, then any program run on that hardware is unreliable. A rare occurrence of a hardware error was discovered in 1994. The widely used Intel processor was found to give incorrect results only when certain numbers were divided, estimated as likely to occur once every 9 billion divisions. Still, the discovery of the error was very big news, and Intel promised to replace the processor for any one that requested it.

The key to developing reliable systems is to keep the design as simple as

possible. In digital computing, all information is represented as a series of digits. We are used to representing numbers using base 10 with digits 0–9. Consider if information were represented within a computer system this way, as shown in Figure 1-9.

4 5 290 1 8

62 4 FIGURE 1-9 Decimal Digitalization

In current electronic computing, each digit is represented by a different voltage level. The more voltage levels (digits) that the hardware must utilize and distinguish, the more complex the hardware design becomes. This results in greater chance of hardware design errors. It is a fact of information theory, however, that any information can be represented using only *two* symbols. Because of this, *all information within a computer system is represented by the use of only two digits, 0 and 1*, called **binary representation**, shown in Figure 1-10.

10 1 0 0 0 1 0 1 1 FIGURE 1-10 Binary Digitalization

In this representation, each digit can be one of only two possible values, similar to a light switch that can be either on or off. Computer hardware, therefore, is based on the use of simple electronic “on/off” switches called **transistors** that switch at very high speed. **Integrated circuits**(“chips”), the building blocks of computer hardware, are comprised of millions or even billions of transistors. The development of the transistor and integrated circuits is discussed in Chapter 12. We discuss binary representation next.

All information within a computer system is represented using only two digits, 0

and 1, called **binary representation.**

1.3.2 The Binary Number System

For representing numbers, any base (radix) can be used. For example, in base 10, there are ten possible digits (0, 1, . . ., 9), in which each column value is a power of ten as shown in Figure 1-11.

10,000,000	1,000,000	100,000	10,000	1,000	100	10	1	
10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0	
						9	**9**	= **99**

FIGURE 1-11 Base 10 Representation

Other radix systems work in a similar manner. **Base 2** has digits 0 and 1, with place values that are powers of two, as depicted in Figure 1-12.

128	64	32	16	8	4	2	1	
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
0	**1**	**1**	**0**	**0**	**0**	**1**	**1**	
0 +	64 +	32 +	0 +	0 +	0 +	2 +	1	= **99**

FIGURE 1-12 Base 2 Representation

As shown in this figure, converting from base 2 to base 10 is simply a matter of adding up the column values that have a 1.

The term **bit** stands for **bi**nary digi **t**. Therefore, every bit has the value 0 or 1. A **byte** is a group of bits operated on as a single unit in a computer system, usually consisting of eight bits. Although values represented in base 2 are significantly longer than those represented in base 10, binary representation is used in digital computing because of the resulting simplicity of hardware design.

The algorithm for the conversion from base 10 to base 2 is to successively divide a number by two until the remainder becomes 0. The remainder of each division provides the next higher-order (binary) digit, as shown in Figure 1-13.

```
99/2  =  49, with remainder 1
49/2  =  24, with remainder 1
24/2  =  12, with remainder 0
12/2  =  6,  with remainder 0
6/2   =  3,  with remainder 0
3/2   =  1,  with remainder 1
1/2   =  0,  with remainder 1
```

FIGURE 1-13
Converting from Base 10 to Base 2

Thus, we get the binary representation of 99 to be 1100011. This is the same as in Figure 1-12 above, except that we had an extra leading insignificant digit of 0, since we used an eight-bit representation there.

The term **bit** stands for binary digit. A **byte** is a group of bits operated on as a single unit in a computer system, usually consisting of eight bits.

1.3.3 Fundamental Hardware Components

The **central processing unit (CPU)** is the “brain” of a computer system, containing digital logic circuitry able to interpret and execute instructions. **Main memory** is where currently executing programs reside, which the CPU can directly and very quickly access. Main memory is volatile; that is, the contents are lost when the power is turned off. In contrast, **secondary memory** is nonvolatile, and therefore provides long-term storage of programs and data. This kind of storage, for example, can be magnetic (hard drive), optical (CD or DVD), or nonvolatile flash memory (such as in a USB drive). **Input/output devices** include anything that allows for input (such as the mouse and keyboard) or output (such as a monitor or printer). Finally, **buses** transfer data between components within a computer system, such as between the CPU and main memory. The relationship of these devices is depicted in Figure 1-14 below.

The **central processing unit (CPU)** is the “brain” of a computer, containing digital logic circuitry able to interpret and execute instructions.

1.3.4 Operating Systems—Bridging Software and Hardware

An **operating system** is software that has the job of managing and interacting with the hardware resources of a computer. Because an operating system is intrinsic to the operation a computer, it is referred to as **system software**.

FIGURE 1-14 Fundamental Hardware Components

An operating system acts as the “middle man” between the hardware and executing application programs (see Figure 1-15). For example, it controls the allocation of memory for the various programs that may be executing on a computer. Operating systems also provide a particular user interface. Thus, it is the operating system installed on a given computer that determines the “look and feel” of the user interface and how the user interacts with the system, and not the particular model computer.

An **operating system** is software that has the job of managing the hardware resources of a given computer and providing a

particular user interface.

FIGURE 1-15 Operating System

1.3.5 Limits of Integrated Circuits Technology: Moore's Law

In 1965, Gordon E. Moore (Figure 1-16), one of the pioneers in the development of integrated circuits and cofounder of Intel Corporation, predicted that as a result of continuing engineering developments, the number of transistors that would be able to be put on a silicon chip would double roughly every two years, allowing the complexity and therefore the capabilities of integrated circuits to grow exponentially. This prediction became known as **Moore's Law**. Amazingly, to this day that prediction has held true. While this doubling of performance cannot go on indefinitely, it has not yet reached its limit.

FIGURE 1-16 Gordon E. Moore
Moore's Law states that the number of transistors that can be placed on a single silicon chip doubles roughly every two years.

Self-Test Questions
1. All information in a computer system is in binary representation. (TRUE/FALSE)
2. Computer hardware is based on the use of electronic switches called _____.

3. How many of these electronic switches can be placed on a single integrated circuit, or "chip"? **(a)** Thousands
(b) Millions
(c) Billions

4. The term "bit" stands for _____.
5. A bit is generally a group of eight bytes. (TRUE/FALSE)

6. What is the value of the binary representation 0110.
(a) 12
(b) 3
(c) 6

7. The _____ interprets and executes instructions in a computer system. **8.** An operating system manages the hardware resources of a computer

system, as well as provides a particular user interface. (TRUE/FALSE)
9. Moore's Law predicts that the number of transistors that can fit on a chip doubles about every ten years. (TRUE/FALSE)
ANSWERS: 1. True, 2. transistors, 3. (c), 4. binary digit, 5. False, 6. (c), 7. CPU, 8. True, 9. False

1.4 Computer Software

The first computer programs ever written were for a mechanical computer designed by Charles Babbage in the mid-1800s. (Babbage's Analytical Engine is discussed in Chapter 12). The person who wrote these programs was a woman, Ada Lovelace (Figure 1-17), who was a talented mathematician. Thus, she is referred to as "the first computer programmer." This section discusses fundamental issues of computer software.

1.4.1 What Is Computer Software?

Computer software is a set of program instructions, including related data and documentation, that can be executed by computer. This can be in the form of instructions on paper, or in digital form. While system software is intrinsic to a computer system, **application software** fulfills users' needs, such as a photo-editing program. We discuss the important concepts of syntax, semantics, and program translation next.

FIGURE 1-17 Ada Lovelace
"The First Computer Programmer"

Computer software is a set of program instructions, including related data and documentation, that can be executed by computer.

1.4.2 Syntax, Semantics, and Program Translation

What Are Syntax and Semantics?

Programming languages (called "artificial languages") are languages just as "natural languages" such as English and Mandarin (Chinese). *Syntax* and *semantics* are important concepts that apply to all languages.

The **syntax** of a language is a set of characters and the acceptable arrangements (sequences) of those characters. English, for example, includes the letters of the alphabet, punctuation, and properly spelled words and properly punctuated sentences. The following is a syntactically correct sentence in English,

"Hello there, how are you?" The following, however, is not syntactically correct, "Hello there, hao are you?"

In this sentence, the sequence of letters “hao” is not a word in the English language. Now consider the following sentence,
“Colorless green ideas sleep furiously.” This sentence is syntactically correct, but is *semantically* incorrect, and thus has no meaning.
1.4 Computer Software 15

The **semantics** of a language is the meaning associated with each syntactically correct sequence of characters. In Mandarin, “Hao” is syntactically correct meaning “good.” (“Hao” is from a system called pinyin, which uses the Roman alphabet rather than Chinese characters for writing Mandarin.) Thus, every language has its own syntax and semantics, as demonstrated in Figure 1-18.

FIGURE 1-18 Syntax and Semantics of Languages

The **syntax** of a language is a set of characters and the acceptable sequences of those characters. The **semantics** of a language is the meaning associated with each syntactically correct sequence of characters.

Program Translation

A central processing unit (CPU) is designed to interpret and execute a specific set of instructions represented in binary form (i.e., 1s and 0s) called **machine code**. Only programs in machine code can be executed by a CPU, depicted in Figure 1-19.

FIGURE 1-19 Execution of Machine Code

Writing programs at this “low level” is tedious and error-prone. Therefore, most programs are written in a “high-level” programming language such as Python. Since the instructions of such programs are not in machine code that a CPU can execute, a translator program must be used. There are two fundamental types of translators. One, called a **compiler**, translates programs directly into machine code to be executed by the CPU, denoted in Figure 1-20.

FIGURE 1-20 Program Execution by Use of a Compiler

The other type of translator is called an **interpreter**, which executes program instructions in place of (“running on top of”) the CPU, denoted in Figure 1-21.

FIGURE 1-21 Program Execution by Use of a Interpreter

Thus, an interpreter can immediately execute instructions as they are entered. This is referred to as **interactive mode**. This is a very useful feature for program development. Python, as we shall see, is executed by an interpreter. On the other hand, compiled programs generally execute faster than interpreted programs. Any program can be executed by either a compiler or an interpreter, as long there exists the corresponding translator program for the programming language that it is written in.

A **compiler** is a translator program that translates programs directly into machine code to be executed by the CPU. An **interpreter** executes program instructions in place of (“running on top of”) the CPU.

Program Debugging: Syntax Errors vs. Semantic Errors

Program debugging is the process of finding and correcting errors (**“bugs”**) in a computer program. Programming errors are inevitable during program development. **Syntax errors** are caused by invalid syntax (for example, entering prnt instead of print). Since a translator cannot understand instructions containing syntax errors, translators terminate when encountering such errors indicating where in the program the problem occurred.

In contrast, **semantic errors** (generally called **logic errors**) are errors in program logic. Such errors cannot be automatically detected, since translators cannot understand the intent of a given computation. For example, if a program computed the average of three numbers as follows,

(num1 1 num2 1 num3) / 2.0

a translator would have no means of determining that the divisor should be 3 and not 2. *Computers do not understand what a program is meant to do, they only follow the instructions given*. It is up to the programmer to detect such errors. Program debugging is not a trivial task, and constitutes much of the time of program development.

Syntax errors are caused by invalid syntax. **Semantic (logic) errors** are caused by errors in program logic.

1.4.3 Procedural vs. Object-Oriented Programming

Programming languages fall into a number of *programming paradigms*. The two major programming paradigms in use today are *procedural (imperative) programming* and *object-oriented programming*. Each provides a different way of thinking about computation. While most programming languages only support one paradigm, Python supports both procedural and object-oriented programming. We will start with the procedural aspects of Python. We then introduce objects in Chapter 6, and delay complete discussion of object-oriented programming until Chapter 10.

Procedural programming and **object-oriented programming** are two major programming paradigms in use today.

Self-T est Questions

1. Two general types of software are system software and _____

software.

2. The syntax of a given language is,
(a) the set of symbols in the language.
(b) the acceptable arrangement of symbols.
(c) both of the above

3. The semantics of a given language is the meaning associated with any arrangement of symbols in the language. (TRUE/FALSE)
4. CPUs can only execute instructions that are in binary form called _____. **5.** The two fundamental types of translation programs for the execution of computer programs are _____ and _____.
6. The process of finding and correcting errors in a computer program is called _____.

7. Which kinds of errors can a translator program detect?
(a) Syntax errors
(b) Semantic errors
(c) Neither of the above

8. Two major programming paradigms in use today are _____ programming and _____ programming.
ANSWERS: 1. application, 2. (c), 3. False, 4. machine code, 5. compilers, interpreters, 6. program debugging, 7. (a), 8. procedural, object-oriented

COMPUTATIONAL PROBLEM SOLVING

1.5 The Process of Computational Problem Solving

Computational problem solving does not simply involve the act of computer programming. It is a *process*, with programming being only one of the steps. Before a program is written, a design for the program must be developed. And before a design can be developed, the problem to be solved must be well understood. Once written, the program must be thoroughly tested. These steps are outlined in Figure 1-22.

FIGURE 1-22 Process of Computational Problem Solving

1.5.1 Problem Analysis

Understanding the Problem

Once a problem is clearly understood, the fundamental computational issues for solving it can be determined. For each of the problems discussed earlier, the representation is straightforward. For the calendar month problem, there are two algorithmic tasks— determining the first day of a given month, and displaying the calendar month in the proper format. The first day of the month can be obtained by *direct calculation* by use of the algorithm provided in Figure 1-8.

For the Man, Cabbage, Goat, Wolf (MCGW) problem, a brute-force algorithmic approach of trying all possible solutions works very well, since there are a small

number of actions that can be taken at each step, and only a relatively small number of steps for reaching a solution. For both the Traveling Salesman problem and the game of chess, the brute-force approach is infeasible. Thus, the computational issue for these problems is to find other, more efficient algorithmic approaches for their solution. (In fact, methods have been developed for solving Traveling Salesman problems involving tens of thousands of cities. And current chess-playing programs can beat top-ranked chess masters.)

San Francisco

Boston 2708 mi.

New York Chicago
344 mi.

748 mi. 1749 mi.

Los Angeles Atlanta

Knowing What Constitutes a Solution

Besides clearly understanding a computational problem, one must know what constitutes a solution. For some problems, there is only one solution. For others, there may be a number (or infinite number) of solutions. Thus, a program may be stated as finding,

♦ *A* solution
♦ An *approximate* solution
♦ A *best*solution
♦ *All*solutions

For the MCGW problem, there are an infinite number of solutions since the man could pointlessly row back and forth across the river an arbitrary number of times. A *best solution* here is one with the shortest number of steps. (There may be more than one "best" solution for any given problem.) In the Traveling Salesman problem there is only one solution (unless there exists more than one shortest route). Finally, for the game of chess, the goal (solution) is to win the game. Thus, since the number of chess games that can be played is on the order of 10^{120} (with each game ending in a win, a loss, or a stalemate), there are a comparable number of possible solutions to this problem.

1.5.2 Program Design
Describing the Data Needed

For the Man, Cabbage, Goat, Wolf problem, a list can be used to represent the correct location (east and west) of the man, cabbage, goat, and wolf as discussed earlier, reproduced below, man cabbage goat wolf
[W, E, W, E]

For the Calendar Month problem, the data include the month and year (entered by the user), the number of days in each month, and the names of the days of the week. A useful structuring of the data is given below,

[month, year]
[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']

The month and year are grouped in a single list since they are naturally associated. Similarly, the names of the days of the week and the number of days

in each month are grouped. (The advantages of list representations will be made clear in Chapter 4.) Finally, the first day of the month, as determined by the algorithm in Figure 1-8, can be represented by a single integer,

0 – Sunday, 1 – Monday, . . ., 6 – Saturday
For the Traveling Salesman problem, the distance between each pair of cities must be represented. One possible way of structuring the data is as a table, depicted in Figure 1-23.
For example, the distance from Atlanta to Los Angeles is 2175 miles. There is duplication of information in this representation, however. For each distance from *x* to *y*, the distance from *y* to *x*is

	Atlanta	Boston	Chicago	Los Angeles	New York City	San Francisco
Atlanta	-	1110	718	2175	888	2473
Boston	1110	-	992	2991	215	3106
Chicago	718	992	-	2015	791	2131
Los Angeles	2175	2991	2015	-	2790	381
New York City	888	215	791	2790	-	2901
San Francisco	2473	3106	2131	381	2901	-

FIGURE 1-23 Table Representation of Data

also represented. If the size of the table is small, as here, this is not much of an issue. However, for a significantly larger table, significantly more memory would be wasted during program execution. Since only half of the table is really needed (for example, the shaded area in the figure), the data could be represented as a list of lists instead,

[[‘Atlanta’, [‘Boston’, 1110], [‘Chicago’, 718], [‘Los Angeles’, 2175], [‘New York’, 888], [‘San Francisco’, 2473]],
[‘Boston’, [‘Chicago’, 992], [‘Los Angeles’, 2991], [‘New York’, 215], [‘San Francisco’, 3106]],
[‘Chicago’, [‘Los Angeles’, 2015], [‘New York’, 791], [‘San Francisco’, 2131]],
[‘Los Angeles’, [‘New York’, 2790], [‘San Francisco’, 381]],
[‘New York’, [‘San Francisco’, 2901]]]

Finally, for a chess-playing program, the location and identification of each chess piece needs to be represented (Figure 1-24). An obvious way to do this is shown on the left below, in which each piece is represented by a single letter

(‘K’ for the king, ‘Q’ for the queen, ‘N’ for the knight, etc.),

R	N	B	Q	K	B	N	R
P	P	P	P	P	P	P	P
P	P	P	P	P	P	P	P
R	N	B	Q	K	B	N	R

4	2	3	4	5	3	2	4
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-4	-2	-3	-4	-5	-3	-2	-4
-1	-1	-1	-1	-1	-1	-1	-1

FIGURE 1-24 Representations of Pieces on a Chess Board

There is a problem with this choice of symbols, however—there is no way to distinguish the white pieces from the black ones. The letters could be modified, for example, PB for a black pawn and PW for a white pawn. While that may be an intuitive representation, it is not the best representation for a program. A better way would be to represent pieces using positive and negative integers as shown on the right of the figure: 1 for a white pawn and 21 for a black pawn; 2 for a white bishop and 22 for a black bishop, and so forth. Various ways of representing chess boards have been developed, each with certain advantages and disadvantages. The appropriate representation of data is a fundamental aspect of computer science.

Describing the Needed Algorithms

When solving a computational problem, either suitable existing algorithms may be found or new algorithms must be developed. For the MCGW problem, there are standard search algorithms that can be used. For the calendar month problem, a day of the week algorithm already exists. For the Traveling Salesman problem, there are various (nontrivial) algorithms that can be utilized, as mentioned, for solving problems with tens of thousands of cities. Finally, for the game of chess, since it is infeasible to look ahead at the final outcomes of every possible move, there are algorithms that make a best guess at which moves to make. Algorithms that work well in general but are not guaranteed to give the correct result for each specific problem are called *heuristic algorithms*.

1.5.3 Program Implementation

Design decisions provide *general* details of the data representation and the

algorithmic approaches for solving a problem. The details, however, do not specify which programming language to use, or how to implement the program. That is a decision for the implementation phase. Since we are programming in Python, the implementation needs to be expressed in a syntactically correct and appropriate way, using the instructions and features available in Python.

1.5.4 Program Testing

As humans, we have abilities that far exceed the capabilities of any machine, such as using commonsense reasoning, or reading the expressions of another person. However, one thing that we are not very good at is dealing with detail, which computer programming demands. Therefore, while we are enticed by the existence of increasingly capable computing devices that unfailingly and speedily execute whatever instructions we give them, writing computer programs is difficult and challenging. As a result, *programming errors are pervasive, persistent and inevitable*. However, the sense of accomplishment of developing software that can be of benefit to hundreds, thousands, or even millions of people can be extremely gratifying. If it were easy to do, the satisfaction would not be as great.

Given this fact, software testing is a crucial part of software development. Testing is done incrementally as a program is being developed, when the program is complete, and when the program needs to be updated. In subsequent chapters, program testing and debugging will be discussed and expanded upon. For now, we provide the following general truisms of software development in Figure 1-25.

1. Programming errors are pervasive, persistent, and inevitable.
2. Software testing is an essential part of software development.
3. Any changes made in correcting a programming error should be fully understood as to why the changes correct the detected error.

FIGURE 1-25 Truisms of Software Development

Truism 1 reflects the fact that programming errors are inevitable and that we must accept it. As a result of truism 1, truism 2 states the essential role of

software testing. Given the inevitability of programming errors, it is important to test a piece of software in a thorough and systematic manner. Finally, truism 3 states the importance of understanding *why* a given change (or set of changes) in a program fixes a specific error. If you make a change to a program that fixes a given problem but you don't know why it did, then you have lost control of the program logic. As a result, you may have corrected one problem, but inadvertently caused other, potentially more serious ones.

Accountants are committed to reconciling balances to the penny. They do not disregard a discrepancy of one cent, for example, even though the difference between the calculated and expected balances is so small. They understand that a small, seemingly insignificant difference can be the result of two (or more) very big discrepancies. For example, there may be an erroneous credit of $1,500.01 and an erroneous debit of $1,500. (The author has experienced such a situation in which the people involved worked all night to find the source of the error.) Determining the source of errors in a program is very much the same. We next look at the Python programming language.

1.6 The Python Programming Language

Now that computational problem solving and computer programming have been discussed, we turn to the Python programming language and associated tools to begin putting this knowledge into practice.

1.6.1 About Python

Guido van Rossum (Figure 1-26) is the creator of the Python programming language, first released in the early 1990s. Its name comes from a 1970s British comedy sketch television show called *Monty Python's Flying Circus*. (Check them out on YouTube!) The development environment IDLE provided with Python (discussed below) comes from the name of a member of the comic group.

Python has a simple syntax. Python programs are clear and easy to read. At the same time, Python provides powerful programming features, and is widely used. Companies and organizations that use Python include YouTube, Google, Yahoo, and NASA. Python is well supported and freely available at www.python.org. (See the Python 3 Programmers' Reference at the end of the text for how to download and install Python.)

1.6.2 The IDLE Python Development Environment

IDLE is an **integrated development environment** (**IDE**). An IDE is a bundled set of software tools

for program development. This typically includes

FIGURE 1-26 Guido van Rossum

an **editor** for creating and modifying programs, a **translator** for executing programs, and a **program debugger**. A debugger provides a means of taking control of the execution of a program to aid in finding program errors.

Python is most commonly translated by use of an interpreter. Thus, Python provides the very useful ability to execute in interactive mode. The window that

provides this interaction is referered to as the **Python shell**. Interacting with the shell is much like using a calculator, except that, instead of being limited to the operations built into a calculator (addition, subtraction, etc.), it allows the entry and creation of any Python code. Example use of the Python shell is demonstrated in Figure 1-27.

FIGURE 1-27 Python Shell
Here, the expression 2 13 is entered at the **shell prompt** (. . .), which immediately responds with the result 5.

Although working in the Python shell is convenient, the entered code is not saved. Thus, for program development, a means of entering, editing, and saving Python programs is provided by the program editor in IDLE. Details are given below.

An **Integrated Development Environment (IDE)** is a bundled set of software tools for program development.

1.6.3 The Python Standard Library

The **Python Standard Library** is a collection of *built-in modules*, each providing specific functionality beyond what is included in the “core” part of Python. (We discuss the creation of Python modules in Chapter 7.) For example, the math module provides additional mathematical functions. The random module provides the ability to generate random numbers, useful in programming, as we shall see. (Other Python modules are described in the Python 3 Programmers’ Reference.) In order to utilize the capabilities of a given module in a specific program, an **import** statement is used as shown in Figure 1-28.

The example in the figure shows the use of the import math statement to gain

access to a particular function in the math module, the factorial function. The syntax for using the factorial

FIGURE 1-28 Using an import statement

function is math.factorial(n), for some positive integer n. We will make use of library modules in Chapter 2. In section 1.7, we see how to enter and execute a complete Python program.

The **Python Standard Library** is a collection of *modules*, each providing specific functionality beyond what is included in the core part of Python.

1.6.4 A Bit of Python

We introduce a bit of Python, just enough to begin writing some simple programs. Since all computer programs input data, process the data, and output results, we look at the notion of a variable, how to perform some simple arithmetic calculations, and how to do simple input and output.

Variables

One of the most fundamental concepts in programming is that of a *variable*. (Variables are discussed in detail in Chapter 2.) A simple description of a variable is “a name that is assigned to a value,” as shown below,

n 55 variable is assigned the value 5 Thus, whenever variable n appears in a calculation, it is the current value that n is assigned to that is used, as in the following,

n 120 (5 1 20) If variable n is assigned a new value, then the same expression will produce a different result,

n 510

n 1 20 (10 1 20) We next look at some basic arithmetic operators of Python. A **variable** is “a name that is assigned to a value.”
Some Basic Arithmetic Operators

The common arithmetic operators in Python are 1 (addition), 2 (subtraction), * (multiplication), / (division), and ** (exponentiation). Addition, subtraction, and division use the same symbols as standard mathematical notation,

10 120 25 215 20 / 10
(There is also the symbol // for truncated division, discussed in Chapter 2.) For multiplication and exponentiation, the asterisk (*) is used.
5 ∗10 (5 times 10) 2 ∗∗4 (2 to the 4th power) Multiplication is never denoted by the use of parentheses as in mathematics, as depicted below, 10 * (20 15) CORRECT 10(20 15) INCORRECT Note that parentheses may be used to denote subexpressions. Finally, we see how to input information from the user, and display program results.
The common arithmetic operators in Python are 1 (addition), 2 (subtraction), * (multiplication), / (division), and ** (exponentiation).
Basic Input and Output
The programs that we will write request and get information from the user. In Python, the input function is used for this purpose,
name 5 input('What is your name?: ') Characters within quotes are called *strings*. This particular use of a string, for requesting input from The input function displays the string on the screen to prompt the user the user, is called a *prompt*. for input,
What is your name?: Charles

The underline is used here to indicate the user’s input.
The print function is used to display information on the screen in Python. This may be used to
display a message,

>>> print('Welcome to My First Program!') Welcome to My First Program!
or used to output the value of a variable,
>>> n 5 10 >>> print(n) 10
or to display a combination of both strings and variables,

>>> name 5 input('What is your name?: ') What is your name?: Charles
>>> print('Hello', name)

Hello Charles

Note that a comma is used to separate the individual items being printed, causing a space to appear between each when displayed. Thus, the output of the print function in this case is Hello Charles, and not HelloCharles. There is more to say about variables, operators, and input/ output in Python. This will be covered in the chapters ahead.

In Python, input is used to request and get information from the user, and print is used to display information on the screen.

1.6.5 Learning How to Use IDLE

In order to become familiar with writing your own Python programs using IDLE, we will create a simple program that asks the user for their name and responds with a greeting. This program utilizes the following concepts:

♦ creating and executing Python programs

♦ input and print

First, to create a Python program file, select New Window from the File menu in the Python shell as shown in Figure 1-29:

FIGURE 1-29 Creating a Python Program File

A new, untitled program window will appear:

Type the following in the program window exactly as shown.

When finished, save the program file by selecting Save As under the File menu, and save in the appropriate folder with the name MyFirstProgram.py.

To run the program, select Run Module from the Run menu (or simply hit function key F5).

If you have entered the program code correctly, the program should execute as shown in Figure 1-30.

FIGURE 1-30 Sample Output of MyFirstProgram.py

If, however, you have mistyped part of the program resulting in a syntax error (such as mistyping print), you will get an error message similar to that in Figure

1-31.

FIGURE 1-31 Output Resulting from a Syntax Error

In that case, go back to the program window and make the needed corrections, then re-save and re-execute the program. You may need to go through this process a number of times until all the syntax errors have been corrected.

1.7 A First Program—Calculating the Drake Equation

Dr. Frank Drake conducted the first search for radio signals from extraterrestrial civilizations in 1960. This established SETI (Search for Extraterrestrial Intelligence), a new area of scientific inquiry. In order to estimate the number of civilizations that may exist in our galaxy that we may be able to communicate with, he developed what is now called the *Drake equation*.

The Drake equation accounts for a number of different factors. The values used for some of these are the result of scientific study, while others are only the result of an “intelligent guess.” The factors consist of R, the average rate of star creation per year in our galaxy; p, the percentage of those stars that have planets; n, the average number of planets that can potentially support life for each star with planets; f, the percentage of those planets that actually go on to develop life; i, the percentage of those planets that go on to develop intelligent life; c, the percentage of those that have the technology communicate with us; and L, the expected lifetime of civilizations (the period that they can communicate). The Drake equation is simply the multiplication of all these factors, giving N, the estimated number of detectable civilizations there are at any given time,

N 5 R ? p ? n ? f ? i ? c ? L

Figure 1-32 shows those parameters in the Drake equation that have some consensus as to their correct value.

Drake Equation Factor Values		Estimated Values
Rate of star creation	R	7†
Percentage of stars with planets	p	40%
Average number of planets that can potentially support life for each star with planets	n	(no consensus)
Percentage of those that go on to develop life	f	13%
Percentage of those that go on to intelligent develop life	i	(no consensus)
Percentage of those willing and able to communicate	c	(no consensus)
Expected lifetime of civilizations	L	(no consensus)
† Estimate of NASA and the European Space Agency		

FIGURE 1-32 Proposed Values for the Drake Equation

1.7.1 The Problem

The value of 7 for R, the rate of star creation, is the least disputed value in the Drake equation today. Given the uncertainty of the remaining factors, you are to develop a program that allows a user to enter their own estimated values for the

remaining six factors (p, n, f, i, c, and L) and displays the calculated result.

1.7.2 Problem Analysis
This problem is very straightforward. We only need to understand the equation provided.
1.7.3 Program Design

The program design for this problem is also straightforward. The data to be represented consist of numerical values, with the Drake equation as the algorithm. The overall steps of the program are depicted in Figure 1-33.

FIGURE 1-33 The Overall Steps of the Drake Equation Program
1.7.4 Program Implementation

The implementation of this program is fairly simple. The only programming elements needed are input, assignment, and print, along with the use of arithmetic operators. An implementation is given in Figure 1-34. Example execution of the program is given in Figure 1-35.

First, note the program lines beginning with the hash sign, #. In Python, this symbol is used to denote a *comment statement*. A **comment statement** contains

information for persons reading the program. Comment statements are ignored during program execution—they have no effect on the program results. In this program, the initial series of comment statements (**lines 1–23**) explain the Drake equation and provide a brief summary of the purpose of the program.

```
 1 # SETI Program
 2 #
 3 # The Drake equation, developed by Frank Drake in the 1960s, attempts to
 4 # estimate how many extraterrestrial civilizations, N, may exist in our
 5 # galaxy at any given time that we might come in contact with,
 6 #
 7 #    N = R * p * n * f * i * c * L
 8 #
 9 # where,
10 #
11 #    R ... estimated rate of star creation in our galaxy
12 #    p ... estimated percent of stars that have planets
13 #    n ... estimated average number of planets that can potentially support
14 #          life for each star with planets
15 #    f ... estimated percent of those planets that actually go on to develop life
16 #    i ... estimated percent of those planets go on to develop intelligent life
17 #    c ... estimated percent of those that are willing and able to communicate
18 #    L ... estimated expected lifetime of such civilizations
19 #
20 # Given that the value for R, 7 per year, is the least disputed of the values,
21 # the user will be prompted to enter estimated values for the remaining six
22 # factors. The estimated number of civilizations that may be detected in our
23 # galaxy will then be displayed.
24
25 # display program welcome
26 print('Welcome to the SETI program')
27 print('This program will allow you to enter specific values related to')
28 print('the likelihood of finding intelligent life in our galaxy. All')
29 print('percentages should be entered as integer values, e.g., 40 and not .40')
30 print()
31
32 # get user input
33 p = int(input('What percentage of stars do you think have planets?: '))
34 n = int(input('How many planets per star do you think can support life?: '))
35 f = int(input('What percentage do you think actually develop life?: '))
36 i = int(input('What percentage of those do you think have intelligent life?: '))
37 c = int(input('What percentage of those do you think can communicate with us?: '))
38 L = int(input('Number of years you think civilizations last?: '))
39
40 # calculate result
41 num_detectable_civilizations = 7 * (p/100) * n * (f/100) * (i/100) * (c/100) * L
42
43 # display result
44 print()
45 print('Based on the values entered ...')
46 print('there are an estimated', round(num_detectable_civilizations),
47        'potentially detectable civilizations in our galaxy')
```

FIGURE 1-34 Drake Equation Program

Comment statements are also used in the program to denote the beginning of each program section (**lines 25, 32, 40,** and **43**). These *section headers* provide a

broad outline of the program following the program design depicted in Figure 1-33.

The program welcome section (**lines 25–30**) contains a series of print instructions displaying output to the screen. Each begins with the word print followed by a matching pair of parentheses. Within the parentheses are the items to be displayed. In this case, each contains a particular string of characters. The final “empty” print, print() on **line 30** (and **line 44**), does not display anything. It simply causes the screen cursor to move down to the next line, therefore creating a skipped line in the screen output. (Later we will see another way of creating the same result.)

```
Program Execution ...

Welcome to the SETI program
This program will allow you to enter specific values related to
the likelihood of finding intelligent life in our galaxy. All
percentages should be entered as integer values, e.g., 40 and not .40

What percentage of stars do you think have planets?: 40
How many planets per star do you think can support life?: 2
What percentage do you think actually develop life?: 5
What percentage of those do you think have intelligent life?: 3
What percentage of those do you think can communicate with us?: 5
Number of years you think civilizations last?: 10000

Based on the values entered...
there are an estimated 4.2 potentially detectable civilizations in
our galaxy
>>>
```

FIGURE 1-35 Execution of the Drake Equation Program

The following section (**lines 32–38**) contains the instructions for requesting the input from the user. Previously, we saw the input function used for inputting a name entered by the user. In that case, the instruction was of the form,

name 5 input('What is your name?:')
In this program, there is added syntax,
p 5 **int(**input('What percentage of stars do you think have planets?:')

The input function always returns what the user enters as a string of characters. This is appropriate when a person’s name is being entered. However, in this program, numbers are being entered, not names. Thus, in, the following,

What percentage of stars do you think have planets?: 40

40 is to be read as single number and not as the characters '4' and '0'. The addition of the int (. . .) syntax around the input instruction accomplishes this. This will be discussed more fully when numeric and string (character) data are discussed in Chapter 2.

On **line 41** the Drake equation is calculated and stored in variable num_detectable_ civilizations. Note that since some of the input values were meant to be percentages (p, f, i, and c), those values in the equation are each divided by 100. Finally, **lines 44–47** display the results.

1.7.5 Program Testing

To test the program, we can calculate the Drake equation for various other values using a calculator, providing a set of *test cases*. A **test case** is a set of input values and expected output of a given program. A **test plan** consists of a number of test cases to verify that a program meets all requirements. A good strategy is to include “average,” as well as “extreme” or “special” cases in a test plan. Such a test plan is given in Figure 1-36.

Based on these results, we can be fairly confident that the program will give the correct results for all input values.

Input Values	p	n	f	i	c	L	Expected Results	Actual Results	Evaluation
Extreme Cases (no chance of contacting intelligent life)									
Zero Planets per Star	0	2	100%	1%	1%	10,000	0	0	passed
Zero percent of Planets Support Life	50%	0	100%	1%	1%	10,000	0	0	passed
Average Cases									
Average Case 1	30%	3	75%	1%	5%	5,000	12	12	passed
Average Case 2	50%	6	80%	5%	10%	10,000	840	840	passed
Extreme Cases (great chance of contacting intelligent life)									
Extreme Case 1	100%	10	100%	100%	100%	10,000	700,000	700,000	passed
Extreme Case 2	100%	12	100%	100%	100%	100,000	8,400,000	8,400,000	passed

FIGURE 1-36 Test Plan Results for Drake's Equation Program

CHAPTER SUMMARY General Topics

Computational Problem Solving/Representation/ Abstraction
Algorithms/Brute Force Approach
Computer Hardware/Transistors/Integrated Circuits/ Moore’s Law
Binary Representation/Bit/Byte/
Binary-Decimal Conversion
Central Processing Unit (CPU)/Main and Secondary Memory
Input/Output Devices/Buses
Computer Software/System Software/
Application Software

Operating Systems
Syntax and Semantics/Program Translation/ Compiler vs. Interpreter
Program Debugging/Syntax Errors vs. Logic (Semantic) Errors
Procedural Programming vs. Object-Oriented Programming
The Process of Computational Problem Solving/ Program Testing
Integrated Development Environments (IDE)/ Program Editors/Debuggers
Comment Statements as Program Documentation
Test Cases/Test Plans

Python-Specific Programming Topics

The Python Programming Language/ Guido van Rossum (creator of Python)
Comment Statements in Python
Introduction to Variables, Arithmetic Operators, input and print in Python
Introduction to Strings with the input Function in Python
Introduction to the Python Standard Library and the import Statement

CHAPTER EXERCISES Section 1.1

The Python Shell/The IDLE Integrated Development Environment
The Standard Python Library

1. Search online for two computing-related fields named “computational X” other than the ones listed in Figure 1-1.
2. Search online for two areas of computer science other than those given in the chapter.
3. For the Man, Cabbage, Goat, Wolf problem:
(a) List all the invalid states for this problem, that is, in which the goat is left alone with the cabbage, or the wolf is left alone with the goat.
(b) Give the shortest sequence of steps that solves the MCGW problem.
(c) Give the sequence of state representations that correspond to your solution starting with (E,E,E,E) and ending with (W,W,W,W).
(d) There is an alternate means of representing states. Rather than a sequence representation, a set representation can be used. In this representation, if an item is on the east side of the river, its symbol is in the set, and if on the west side, the symbol is not in the set as shown below,

{M,C,G,W}—all items on east side of river (start state)
{C,W}—cabbage and wolf on east side of river, man and goat on west side { }

—all items on the west side of the river (goal state)

Give the sequence of states for your solution to the problem using this new state representation. **(e)** How many shortest solutions are there for this problem?

4. For a simple game that starts with five stones, in which each player can pick up either one or two stones, the person picking up the last stone being the loser,
(a) Give a state representation appropriate for this problem.
(b) Give the start state and goal state for this problem.
(c) Give a sequence of states in which the first player wins the game.

Section 1.2
5. Using the algorithm in Figure 1-8, *show all steps* for determining the day of the week for January 24,

2018. (Note that 2018 is not a leap year.)
6. Using the algorithm in Figure 1-8, determine the day of the week that you were born on.

7. Suppose that an algorithm was needed for determining the day of the week for dates that only occur within the years 2000–2099. Simplify the day of the week algorithm in Figure 1-8 as much as possible by making the appropriate changes.

8. As precisely as possible, give a series of steps (an algorithm) for doing long addition.
Section 1.3
9. What is the number of bits in 8 bytes, assuming the usual number of bits in a byte? **10.** Convert the following values in binary representation to base 10. *Show all steps*. **(a)** 1010 **(b)** 1011 **(c)** 10000 **(d)** 1111
Chapter Exercises 35

11. Convert the following values into binary (base 2) representation. *Show all steps*.
(a) 5 **(b)** 7 **(c)** 16 **(d)** 15 **(e)** 32
(f) 33 **(g)** 64 **(h)** 63 **(i)** 128 **(j)** 127

12. What is in common within each of the following groups of binary numbers?
(a) values that end with a “0” digit (e.g., 1100)
(b) values that end with a “1” digit (e.g., 1101)
(c) values with a leftmost digit of “1,” followed by all “0s” (e.g., 1000)

(d) values consisting only of all “1” digits (e.g., 1111)

13. Assuming that Moore’s Law continues to hold true, where n is the number of transistors that can currently be placed on an integrated circuit (chip), and k*n is the number that can be placed on a chip in eight years, what is the value of k?

Section 1.4

14. Give two specific examples of an application program besides those mentioned in the chapter.

15. For each of the following statements in English, indicate whether the statement contains a syntax error, a logic (semantic) error, or is a valid statement.

(a) Witch way did he go?

(b) I think he went over their.

(c) I didn’t see him go nowhere.

16. For each of the following arithmetic expressions for adding up the integers 1 to 5, indicate whether the expression contains a syntax error, a semantic error, or is a valid expression.

(a) 1 12 113 14 15

(b)1 12 14 15

(c) 1 12 13 14 15

(d)5 14 1 3 12 11

17. Give one benefit of the use of a compiler, and one benefit of the use of an interpreter.

Section 1.5

18. Use the Python Interactive Shell to calculate the number of routes that can be taken for the Traveling Salesman problem for:

(a) 6 cities **(b)** 12 cities **(c)** 18 cities **(d)** 36 cities

19. Enter the following statement into the interactive shell:

print('What is your favorite color?')

Record the output. Now enter the following statement exactly as given,

printt('What is your favorite color?')

Record the output. Is this a syntax error or a logic error?

20. For the Traveling Salesman problem,

(a) Update the list representation of the distances between cities in the table in Figure 1-23 to add the city of Seattle. The distances between Seattle and each of

the other cities is given below.

Atlanta to Seattle, 2641 miles, Boston to Seattle, 3032 miles, Chicago to Seattle, 2043 miles, LA to Seattle, 1208 miles, NYC to Seattle, 2832 miles, San Francisco to Seattle, 808 miles

(b) Determine a reasonably short route of travel for visiting each city once and only once, starting in Atlanta and ending in San Francisco.
Section 1.6

21. Which of the following capabilities does an integrated development environment (IDE) provide? **(a)** Creating and modifying programs
(b) Executing programs
(c) Debugging programs
(d) All of the above

22. The Python shell is a window in which Python instructions are immediately executed. (TRUE/FALSE)
23. Suppose that the math module of the Python Standard Library were imported. What would be the proper syntax for calling a function in the math module named sqrt to calculate the square root of four?
24. What is the value of variable n after the following instructions are executed?
j 5 5
k 5 10
n 5 j * k

25. Which of the following is a proper arithmetic expression in Python?
(a) 10(15 1 6)
(b)(10 * 2)(4 1 8)
(c) 5 * (6 - 2)

26. Exactly what is output by the following if the user enters 24 in response to the input prompt. age 5 input('How old are you?: ')
print('You are', age, 'years old')
PYTHON PROGRAMMING EXERCISES

P1. Write a simple Python program that displays the following powers of 2, one per line: 2 2 2 2 2 2 2 2
P2. Write a Python program that allows the user to enter any integer value, and displays the value of 2 raised to that power. Your program should function as

shown below.

What power of two? 10
Two to the power of 10 is 1024
P3. Write a Python program that allows the user to enter any integer base and integer exponent, and displays the value of the base raised to that exponent. Your program should function as shown below.

What base? 10
What power of 10 ? 4
10 to the power of 4 is 10000

P4. Write a Python program that allows the user to enter a four-digit binary number and displays its value in base 10. *Each binary digit should be entered one per line, starting with the leftmost digit*, as shown below.

Enter leftmost digit: 1
Enter the next digit: 0
Enter the next digit: 0
Enter the next digit: 1
The value is 9

Program Development Problems 37

P5. Write a simple Python program that prompts the user for a certain number of cities for the Traveling Salesman problem, and displays the total number of possible routes that can be taken. Your program should function as shown below.

How many cities? 10
For 10 cities, there are 3628800 possible routes
PROGRAM MODIFICATION PROBLEMS **M1.** Modify the sample “hello” Python program in section 1.6.5 to first request the user’s first name, and then request their last name. The program should then display,
Hello *firstname lastname*
Welcome to Python!
for the *firstname* and *lastname* entered.

M2. Modify the Drake’s Equation Program in section 1.7 so that it calculates results for a best case scenario, that is, so that factors p (percentage of stars that have planets), f (percentage of those planets that develop life), i (percentage of

those planets that develop intelligent life), and c (percentage of those planets that can communicate with us) are all hard-coded as 100%. The value of R should remain as 7. Design the program so that the only values that the user is prompted for are how many planets per star can support life, n, and the estimated number of years civilizations last, L. Develop a set of test cases for your program with the included test results.

PROGRAM DEVELOPMENT PROBLEMS

D1. Develop and test a program that allows the user to enter an integer value indicating the number of cities to solve for the Traveling Salesman problem. The program should then output the number of years it would take to solve using a brute force-approach. Make use of the factorial function of the math module as shown in Figure 1-28. Estimate the total amount of time it takes by using the assumptions given in section 1.1.2.

D2. Based on the information provided about the game of chess in section 1.1.2, develop and test a program that determines how many years it would take for all possible chess games to be played if everyone in the world (regardless of age) played one (unique) chess game a day. Assume the current world population to be 7 billion.

CHAPTER 2 Data and Expressions

With this chapter, we begin a detailed discussion of the concepts and techniques of computer programming. We start by looking at issues related to the representation, manipulation, and input/ output of data—fundamental to all computing.

OBJECTIVES

After reading this chapter and completing the exercises, you will be able to: ♦ Explain and use numeric and string literal values

♦ Explain the limitations in the representation of floating-point values ♦ Explain what a character-encoding scheme is

♦ Explain what a control character is

♦ Explain and use variables, identifiers, and keywords

♦ Describe and perform variable assignment

♦ Describe and use operators and expressions

♦ Describe and use operator precedence and operator associativity ♦ Define a

data type, and explain type coercion vs. type conversion ♦ Explain the difference between static and dynamic typing ♦ Effectively use arithmetic expressions in Python
♦ Write a simple straight-line Python program
♦ Explain the importance and use of test cases in program testing

CHAPTER CONTENTS Motivation
Fundamental Concepts 2.1 Literals
2.2 Variables and Identifiers 2.3 Operators

2.4 Expressions and Data Types Computational Problem Solving
2.5 Age in Seconds Program

MOTIVATION

The generation, collection, and analysis of data is a driving force in today's world. The sheer amount of data being created is staggering. Chain stores generate *terabytes* (see Figure 2-1) of customer information, looking for shopping patterns of individuals. Facebook users have created 40 billion photos requiring more than a *petabyte* of storage. A certain radio telescope is expected to generate an *exabyte* of information every four hours. All told, the current amount of data created each year is estimated to be almost two *zettabytes*, more than doubling every two years. In this chapter, we look at how data is represented and operated on in Python.

Term†	Size (bytes)		Equivalent Storage
Kilobyte (KB) Kibibyte	10^3 2^{10}	1,000 1,024	Typewritten page†† (2 KB)
Megabyte (MB) Mebibyte	10^6 2^{20}	1,000,000 1,048,576	A small novel†† (1 MB)
Gigabyte (GB) Gibibyte	10^9 2^{30}	1,000,000,000 1,073,741,824	A pickup truck load of books†† (1 GB)
Terabyte (TB) Tibibyte	10^{12} 2^{40}	1,000,000,000,000 1,099,511,627,776	An academic research library†† (2 TB)
Petabyte (PB) Pebibyte	10^{15} 2^{50}	1,000,000,000,000,000 1,125,899,906,842,624	All U.S. academic libraries†† (2 PB)
Exabyte (EB) Exbibyte	10^{18} 2^{60}	1,000,000,000,000,000,000 1,152,921,504,606,846,976	All words ever spoken†† (5 EB)
Zettabyte (ZB) Zebibyte	10^{21} 2^{70}	1,000,000,000,000,000,000,000 1,180,591,620,717,411,303,424	Amount of data produced each year†††

† *Because of inconsistencies in the definition of Megabyte, Gigabyte, etc., the International Organization for Standards (ISO) has recommended the use of the terms Kilobyte, Megabyte, Gigabyte, etc. for* 10^3, 10^6, 10^9 *etc., and Kibibyte, Mebibyte and Gibibyte for* 2^{10}, 2^{20}, 2^{30}, *etc.* †† *School of Information Management and Systems, University of California Berkeley http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/*
††† *The fifth annual IDC Digital Universe study, 2011 http://bit.ly/lbCBCJ*

FIGURE 2-1 Measurements of Data Size (bytes)

FUNDAMENTAL CONCEPTS

2.1 Literals

2.1.1 What Is a Literal?

To take something literally is to take it at “face value.” The same is true of *literals* in programming. A **literal** is a sequence of one of more characters that stands for itself, such as the literal 12. We look at numeric literals in Python next.

A **literal** is a sequence of one or more characters that stands for itself.

2.1.2 Numeric Literals

A **numeric literal** is a literal containing only the digits 0–9, an optional sign character (1 or 2), and a possible decimal point. (The letter *e* is also used in exponential notation, shown in the next subsection). If a numeric literal contains a decimal point, then it denotes a **floating-point value**, or “ **float**” (e.g., 10.24); otherwise, it denotes an **integer value** (e.g., 10). *Commas are never used in*

numeric literals. Figure 2-2 gives additional examples of numeric literals in Python.

Numeric Literals integer values	floating-point values	incorrect
5	5. 5.0 5.125 0.0005 5000.125	5,000.125
2500	2500. 2500.0 2500.125	2,500 2,500.125
+2500	+2500. +2500.0 +2500.125	+2,500 +2,500.125
-2500	-2500. -2500.0 -2500.125	-2,500 -2,500.125

FIGURE 2-2 Numeric Literals in Python

Since numeric literals without a provided sign character denote positive values, an explicit positive sign character is rarely used. Next we look at how numeric values are represented in a computer system.

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

... 1024 ... 21024 ... .1024 ??? ??? ???
... 1,024 ... 0.1024 ... 1,024.46 ??? ??? ???

A **numeric literal** is a literal containing only the digits 0–9, a sign character (1 or 2) and a possible decimal point. Commas are never used in numeric literals.

Limits of Range in Floating-Point Representation

There is no limit to the size of an integer that can be represented in Python. Floating-point values, however, have both a limited *range* and a limited *precision*. Python uses a double-precision standard format (IEEE 754) providing a range of 10^{2308} to 10^{308} with 16 to 17 digits of precision. To denote such a range of values, floating-points can be represented in scientific notation,

9.0045602e 15 (9.0045602 , 8 digits of precision)
1.006249505236801e8 (1.006249505236801 , 16 digits of precision)
4.239e216 (4.239 , 4 digits of precision)

It is important to understand the limitations of floating-point representation. For example, the multiplication of two values may result in **arithmetic overflow**, a condition that occurs when a calculated result is too large in magnitude (size) to

be represented,

... 1.5e200 * 2.0e210
... inf

This results in the special value inf (“infinity”) rather than the arithmetically correct result 3.0e410, indicating that arithmetic overflow has occurred. Similarly, the division of two numbers may result in **arithmetic underflow**, a condition that occurs when a calculated result is too small in magnitude to be represented,

... 1.0e2300 / 1.0e100 0.0

This results in 0.0 rather than the arithmetically correct result 1.0e2400, indicating that arithmetic underflow has occurred. We next look at possible effects resulting from the limited precision in floating-point representation.

LET’S TRY IT
From the Python Shell, enter the following and observe the results.

... 1.2e200 * 2.4e100 ... 1.2e200 / 2.4e100 ??? ???
... 1.2e200 * 2.4e200 ... 1.2e2200 / 2.4e200 ??? ???
Arithmetic overflow occurs when a calculated result is too large in magnitude to be represented. **Arithmetic underflow** occurs when a calculated result is too small in magnitude to be represented.
Limits of Precision in Floating-Point Representation

Arithmetic overflow and arithmetic underflow are relatively easily detected. The loss of precision that can result in a calculated result, however, is a much more subtle issue. For example, 1/3 is equal to the infinitely repeating decimal .33333333 . . ., which also has repeating digits in base two, .010101010. . . . Since any floating-point representation necessarily contains only a finite number of digits, what is stored for many floating-point values is only an *approximation* of the true value, as can be demonstrated in Python,

... 1/3
.3333333333333333
Here, the repeating decimal ends after the 16th digit. Consider, therefore, the following,
... 3 * (1/3) 1.0

Given the value of 1/3 above, we would expect the result to be .9999999999999999, so what is happening here? The answer is that Python displays a *rounded* result to keep the number of digits displayed manageable. However, the representation of 1/3 as .3333333333333333 remains the same, as demonstrated by the following,

... 1/3 1 1/3 1 1/3 1 1/3 1 1/3 1 1/3 1.9999999999999998

In this case we get a result that reflects the representation of 1/3 as an approximation, since the last digit is 8, and not 9. However, if we use multiplication instead, we again get the rounded value displayed,

... 6 * (1/3) 2.0

The bottom line, therefore, is that no matter how Python chooses to display calculated results, the value stored is limited in both the range of numbers that can be represented and the degree of precision. For most everyday applications, this slight loss in accuracy is of no practical concern. However, in scientific computing and other applications in which precise calculations are required, this is something that the programmer must be keenly aware of.

LET'S TRY IT From the Python Shell, enter the following and observe the results.

... 1/10 ... 6 * (1/10) ??? ???
... 1/10 1 1/10 1 1/10 ... 6 * 1/10 ??? ???
... 10 * (1/10)
???
Since any floating-point representation contains only a finite number of digits, what is stored for many floating-point values is only an *approximation* of the true value.
Built-in **format** Function

Because floating-point values may contain an arbitrary number of decimal places, the built-in **format**function can be used to produce a numeric string version of the value containing a specific number of decimal places,

... 12/5 ... 5/7
2.4 0.7142857142857143 ... format(12/5, '.2f') ... format(5/7, '.2f')
'2.40' '0.71'

In these examples, *format specifier*'.2f' rounds the result to two decimal places of accuracy in the string produced. For very large (or very small) values 'e' can be used as a format specifier,
... format(2 ** 100, '.6e') '1.267651e 130'

In this case, the value is formatted in scientific notation, with six decimal places of precision. Formatted numeric string values are useful when displaying results in which only a certain number of decimal places need to be displayed,

without use of format specifier

... tax 5 0.08
... print('Your cost: $', (1 1 tax) * 12.99) Your cost: $ 14.029200000000001

with use of
format specifier
... print('Your cost: $', format((1 1 tax) * 12.99, '.2f')) Your cost: $ 14.03
Finally, a comma in the format specifier adds comma separators to the result,
... format(13402.25, ' , .2f') 13,402.24
We will next see the use of format specifiers for formatting string values as well.
LET'S TRY IT From the Python Shell, enter the following and observe the results. ... format(11/12, '.2f') ... format(11/12, '.2e') ??? ???
... format(11/12, '.3f') ... format(11/12, '.3e') ??? ???
The built-in **format** function can be used to produce a numeric string of a given floating-point value *rounded* to a specific number of decimal places.
2.1.3 String Literals
Numerical values are not the only literal values in programming. **String literals**, or " **strings**," represent a sequence of characters,
'Hello' 'Smith, John' "Baltimore, Maryland 21210"

In Python, string literals may be *delimited* (surrounded) by a matching pair of either single (') or double (") quotes. Strings must be contained all on one line (except when delimited by triple quotes, discussed in Chapter 7). We have already seen the use of strings in Chapter 1 for displaying screen output,

... print('Welcome to Python!') Welcome to Python!
Additional examples of string literals are given in Figure 2-3.

'A'	- a string consisting of a single character
'jsmith16@mycollege.edu'	- a string containing non-letter characters
"Jennifer Smith's Friend"	- a string containing a single quote character
' '	- a string containing a single blank character
''	- the empty string

FIGURE 2-3 String Literal Values

As shown in the figure, a string may contain zero or more characters, including letters, digits, special characters, and blanks. A string consisting of only a pair of matching quotes (with nothing in between) is called the **empty string**, which is different from a string containing only blank characters. Both blank strings and the empty string have their uses, as we will see. Strings may also contain quote characters as long as different quotes are used to delimit the string,

"Jennifer Smith's Friend"
If this string were delimited with single quotes, the apostrophe (single quote) would be considered the matching closing quote of the opening quote, leaving the last final quote unmatched, 'Jennifer Smith's Friend' … matching quote?

Thus, Python allows the use of more than one type of quote for such situations. (The convention used in the text will be to use single quotes for delimiting strings, and only use double quotes when needed.)

LET’S TRY IT From the Python Shell, enter the following and observe the results.
... print('Hello') ... print('Hello") ... print('Let's Go') ??? ??? ???
... print("Hello") ... print("Let's Go!') ... print("Let's go!") ??? ??? ???
A **string literal**, or **string**, is a sequence of characters denoted by a pair of matching single or double (and sometimes triple) quotes in Python.
The Representation of Character Values

There needs to be a way to encode (represent) characters within a computer. Although various encoding schemes have been developed, the **Unicode** encoding scheme is intended to be a universal encoding scheme. Unicode is actually a collection of different encoding schemes utilizing between 8 and 32 bits for each character. The default encoding in Python uses **UTF-8**, an 8-bit encoding compatible with ASCII, an older, still widely used encoding scheme.

Currently, there are over 100,000 Unicode-defined characters for many of the languages around the world. Unicode is capable of defining more than 4 billion characters. Thus, all the world's languages, both past and present, can potentially be encoded within Unicode. A partial listing of the ASCII-compatible UTF-8 encoding scheme is given in Figure 2-4.

Space	00100000	32	A	01000001	65
!	00100001	33	B	01000010	66
"	00100010	34	C	01000011	67
#	00100011	35	.		
.			.		
.			Z	01011010	90
0	00110000	48	a	01100001	97
1	00110001	49	b	01100010	98
2	00110010	50	c	01100011	99
.			.		
.			.		
9	00111001	57	z	01111010	122

FIGURE 2-4 Partial UTF-8 (ASCII) Code Table

UTF-8 encodes characters that have an ordering with sequential numerical values. For example, 'A' is encoded as 01000001 (65), 'B' is encoded as 01000010 (66), and so on. This is true for character digits as well, '0' is encoded as 00110000 (48) and '1' is encoded as 00110001 (49). This underscores the difference between a numeric representation (that can be used in arithmetic calculations) vs. a number represented as a string of digit characters (that cannot), as demonstrated in Figure 2-5.

FIGURE 2-5 Numeric vs. String Representation of Digits

Python has means for converting between a character and its encoding. The ord function gives the UTF-8 (ASCII) encoding of a given character. For example,

ord('A') is 65. The chr function gives the character for a given encoding value, thus chr(65) is 'A'. (Functions are discussed in Chapter 5.) While in general there is no need to know the specific encoding of a given character, there are times when such knowledge can be useful.
LET'S TRY IT

From the Python Shell, enter the following and observe the results.
... ord('1') ... chr(65) ... chr(97) ??? ??? ??? ... ord('2') ... chr(90) ... chr(122) ??? ??? ???

Unicode is capable of representing over 4 billion different characters, enough to represent the characters of all languages, past and present. Python's (default) character encoding uses **UTF-8**, an eight-bit encoding that is part of the Unicode standard.

2.1.4 Control Characters

Control characters are special characters that are not displayed on the screen. Rather, they *control* the display of output (among other things). Control characters do not have a corresponding keyboard character. Therefore, they are represented by a combination of characters called an *escape sequence*.

An **escape sequence** begins with an **escape character** that causes the sequence of characters following it to "escape" their normal meaning. The backslash (\) serves as the escape character in Python. For example, the escape sequence '\n', represents the *newline control character* , used to begin a new screen line. An example of its use is given below,

print('Hello\nJennifer Smith')
which is displayed as follows,
Hello
Jennifer Smith
Further discussion of control characters is given in the Python 3 Programmers' Reference.
LET'S TRY IT From the Python Shell, enter the following and observe the results.
... print('Hello World') ... print('Hello\nWorld') ??? ???
... print('Hello World\n') ... print('Hello\n\nWorld') ??? ???
... print('Hello World\n\n') ... print(1, '\n', 2, '\n', 3) ??? ???

... print('\nHello World') ... print('\n', 1, '\n', 2, '\n', 3) ??? ???

Control characters are nonprinting characters used to *control* the display of output (among other things). An **escape sequence** is a string of one or more characters used to denote control characters.

2.1.5 String Formatting

We saw above the use of built-in function format for controlling how numerical values are displayed. We now look at how the format function can be used to control how strings are displayed. As given above, the format function has the form,

format(*value, format_specifier)*

where *value* is the value to be displayed, and *format_specifier* can contain a combination of formatting options. For example, to produce the string 'Hello' left-justified in a field width of 20 characters would be done as follows,

format('Hello', ' <20') → 'Hello '
To right-justify the string, the following would be used,
format('Hello', ' >20') → ' Hello'

Formatted strings are left-justified by default. To center the string the '^' character is used: format('Hello', '^20'). Another use of the format function is to create strings of blank characters, which is sometimes useful,

format(' ', '30') → ' '
Finally blanks, by default, are the *fill character* for formatted strings. However, a specific fill character can be specified as shown below,
... print('Hello World', format('.', '. <30'), 'Have a Nice Day!') Hello World
..... Have a Nice Day!
LET'S TRY IT From the Python Shell, enter the following and observe the results.
... print(format('Hello World', '^40'))
???
... print(format('2','2<20'), 'Hello World', format('2','2>20')) ???
Built-in function **format** can be used to control how strings are displayed.
2.1.6 Implicit and Explicit Line Joining

Sometimes a program line may be too long to fit in the Python-recommended maximum length of 79 characters. There are two ways in Python to do deal with such situations—implicit and explicit line joining. We discuss this next.

Implicit Line Joining

There are certain delimiting characters that allow a *logical* program line to span more than one physical line. This includes matching parentheses, square brackets, curly braces, and triple quotes. For example, the following two program lines are treated as one logical line,

```
print('Name:', student_name, 'Address:', student_address, 'Number of Credits:',
total_credits, 'GPA:', current_gpa)
```

Matching quotes (except for triple quotes, covered later) must be on the same physical line. For example, the following will generate an error,

```
print('This program will calculate a restaurant tab for a couple with a gift
certificate, and a restaurant tax of 3%')
```

We will use this aspect of Python throughout the book.

Matching parentheses, square brackets, and curly braces can be used to span a *logical* program line on more than one physical line.

Explicit Line Joining

In addition to implicit line joining, program lines may be explicitly joined by use of the backslash (\) character. Program lines that end with a backslash that are not part of a literal string (that is, within quotes) continue on the following line,

```
numsecs_1900_dob 5 ((year_birth 2 1900) * avg_numsecs_year) 1 \
((month_birth 2 1) * avg_numsecs_month) 1 \ (day_birth * numsecs_day)
```

Program lines may be explicitly joined by use of the backslash (\).

2.1.7 Let’s Apply It—“Hello World Unicode Encoding”

It is a long tradition in computer science to demonstrate a program that simply displays “Hello World!” as an example of the simplest program possible in a particular programming language. In Python, the complete Hello World program is comprised of one program line,

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

We take a twist on this tradition and give a Python program that displays the

Unicode encoding for each of the characters in the string “Hello World!” instead. This program utilizes the following programming features:

➤ string literals ➤ print ➤ord function
The program and program execution are given in Figure 2-6.

Program Execution ...

```
The Unicode encoding for 'Hello World!' is:
72 101 108 108 111 32 87 111 114 108 100 33
```

```
1 # This program displays the Unicode encoding for 'Hello World!'
2
3 # Program greeting
4 print("The Unicode encoding for 'Hello World!' is:")
5
6 # output results
7 print(ord('H'), ord('e'), ord('l'), ord('l'), ord('o'), ord(' '),
8       ord('W'), ord('o'), ord('r'), ord('l'), ord('d'), ord('!'))
```

FIGURE 2-6 Hello World Unicode Encoding Program

The statements on **lines 1** , **3**, and **6** are comment statements, introduced in Chapter 1. They are ignored during program execution, used to provide information to those reading the program. The print function on **line 4** displays the message ‘Hello World!’. Double quotes are used to delimit the corresponding string, since the single quotes within it are to be taken literally. The use of print on **line 7** prints out the Unicode encoding, one-by-one, for each of the characters in the “Hello World!” string. Note from the program execution that there is a Unicode encoding for the blank character (32), as well as the exclamation mark (33).

Self-Test Questions
1. Indicate which of the following are valid numeric literals in Python.
(a)1024 **(b)**1,024 **(c)**1024.0 **(d)**0.25 **(e)**.45 **(f)**0.25 110

2. Indicate which of the following exceed the range and/or precision of floating-point values that can be represented in Python.
(a)1.89345348392e 1301 **(c)**2.0424e2320
(b)1.62123432632322e1300 **(d)**1.3232324353423278964 52e2140

3. Which of the following would result in either overflow or underflow for the

floating-point representation scheme mentioned in the chapter.
(a)6.25e1240 * 1.24e110 **(c)**6.25e 1240 / 1.24e110
(b)2.24e1240 * 1.45e 1300 **(d)**2.24e2240 / 1.45e1300

4. Exactly what is output by print(format(24.893952, '.3f'))
(a)24.894 **(b)**24.893 **(c)**2.48e1
5. Which of the following are valid string literals in Python.
(a)"Hello" **(b)**'hello' **(c)**"Hello' **(d)**'Hello there' **(e)**" **6.** Which of the following results of the ord and chr functions are correct? **(a)**ord('1')→49 **(b)**chr(68) →'d' **(c)**chr(99) →'c' **7.** How many lines of screen output is displayed by the following,
print('apple\nbanana\ncherry\npeach') **(a)**1 **(b)**2 **(c)**3 **(d)**4
ANSWERS: 1. (a,c,d,e), 2. (c), 3. (b, overflow), (d, underflow), 4. (a) 5. (a,b,d,e), 6. (a,c), 7. (d)

2.2 Variables and Identifiers

So far, we have only looked at literal values in programs. However, the true usefulness of a computer program is the ability to operate on *different* values each time the program is executed. This is provided by the notion of a *variable* . We look at variables and identifiers next.

2.2.1 What Is a Variable?

A **variable** is a name (identifier) that is associated with a value, as for variable num depicted in Figure 2-7.

FIGURE 2-7 Program Variable

A variable can be assigned different values during a program's execution—hence, the name "variable." Wherever a variable appears in a program (except on the left-hand side of an assignment statement), *it is the value associated with the variable that is used*, and not the variable's name,

num 1 1 → 10 1 1 → 11

Variables are assigned values by use of the **assignment operator** , 5,
num 510 num 5 num 1 1

Assignment statements often look wrong to novice programmers. Mathematically, num 5 num 1 1 does not make sense. In computing, however, it is used to increment the value of a given variable by one. It is more appropriate,

therefore, to think of the 5 symbol as an arrow symbol, as shown in Figure 2-8.

FIGURE 2-8 Variable Update

When thought of this way, it makes clear that *the right side of an assignment is evaluated first, then the result is assigned to the variable on the left*. An arrow symbol is not used simply because there is no such character on a standard computer keyboard. Variables may also be assigned to the value of another variable (or expression, discussed below) as depicted in Figure 2-9.

FIGURE 2-9 Variable Assignment (to another variable)
Variables num and k are both associated with the same literal value 10 in memory. One way to see this is by use of built-in function id,
... id(num) ... id(k)
505494040 505494040

The id function produces a unique number identifying a specific value (object) in memory. Since variables are meant to be distinct, it would appear that this sharing of values would cause problems. Specifically, if the value of num changed, would variable k change along with it? This cannot happen in this case because the variables refer to integer values, and integer values are *immutable*. An **immutable value** is a value that cannot be changed. Thus, both will continue to refer to the same value until one (or both) of them is reassigned, as depicted in Figure 2-10.

FIGURE 2-10 Variable Reassignment

If no other variable references the memory location of the original value, the memory location is *deallocated* (that is, it is made available for reuse).

Finally, in Python the same variable can be associated with values of different type during program execution, as indicated below.

var 5 12 integer
var 5 12.45 float
var 5 'Hello' string

LET'S TRY IT From the Python Shell, enter the following and observe the results.

... num 510 ...k 530 ...num ...k
??? ???
...id(num) ...num ??? ???

... id(k) ...num 520 ???
...num ...id(num) ??? ???
...id(num)
??? ... k 5 k 1 1

... k ... k 5 num ???
... k ... id(num) ??? ???
... id(k) ... id(k) ??? ???
... id(num)
???

A **variable** is a name that is associated with a value. The **assignment operator**, 5, is used to assign values to variables. An **immutable value** is a value that cannot be changed.

2.2.2 Variable Assignment and Keyboard Input

The value that is assigned to a given variable does not have to be specified in the program, as demonstrated in previous examples. The value can come from the user by use of the input function introduced in Chapter 1,

... name 5 input('What is your first name?') What is your first name? John
In this case, the variable name is assigned the string 'John'. If the user hit return without entering any value, name would be assigned to the empty string (').

All input is returned by the input function as a string type. *For the input of numeric values, the response must be converted to the appropriate type.* Python provides built-in **type conversion functions int**() and **float**() for this purpose, as shown below for a gpa calculation program,

line 5 input('How many credits do you have?') num_credits 5 int(line)
line 5 input('What is your grade point average?') gpa 5 float(line)

Here, the entered number of credits, say '24', is converted to the equivalent integer value, 24, before being assigned to variable num_credits. For input of the gpa, the entered value, say '3.2', is converted to the equivalent floating-point value, 3.2. Note that the program lines above could be combined as follows,

num_credits 5 int(input('How many credits do you have? ')) gpa 5 float(input('What is your grade point average? '))
LET'S TRY IT From the Python Shell, enter the following and observe the results. ... num 5 input('Enter number: ')

Enter number: 5
... num 5 input('Enter name: ') Enter name: John

??? ???

... num 5 int(input('Enter number: ')) ... num 5 int(input('Enter name: ')) Enter number: 5 Enter name: John
??? ???

All input is returned by the input function as a string type. Built-in functions int() and float() can be used to convert a string to a numeric type.
2.2.3 What Is an Identifier?

An **identifier** is a sequence of one or more characters used to provide a name for

a given program element. Variable names line, num_credits, and gpa are each identifiers. Python is *case sensitive*, thus, Line is different from line. Identifiers may contain letters and digits, but cannot begin with a digit. The underscore character, _, is also allowed to aid in the readability of long identifier names. It should not be used as the *fi r s t* character, however, as identifiers beginning with an underscore have special meaning in Python.

Spaces are not allowed as part of an identifier. This is a common error since some operating systems allow spaces within file names. In programming languages, however, spaces are used to delimit (separate) distinct syntactic entities. Thus, any identifier containing a space character would be considered two separate identifiers. Examples of valid and invalid identifiers in Python are given in Figure 2-11.

Valid Identifiers	Invalid Identifiers	Reason Invalid
totalSales	'totalSales'	quotes not allowed
totalsales	total sales	spaces not allowed
salesFor2010	2010Sales	cannot begin with a digit
sales_for_2010	_2010Sales	should not begin with an underscore

FIGURE 2-11 Identifier Naming

LET'S TRY IT From the Python Shell, enter the following and observe the results.

... spring2014SemCredits 5 15 ... spring2014-sem-credits 5 15 ??? ???

... spring2014_sem_credits 5 15 ... 2014SpringSemesterCredits 5 15 ??? ???

An **identifier** is a sequence of one or more characters used to name a given program element. In Python, an identifier may contain letters and digits, but cannot begin with a digit. The special underscore character can also be used.

2.2.4 Keywords and Other Predefined Identifiers in Python

A **keyword** is an identifier that has predefined meaning in a programming language. Therefore, keywords cannot be used as "regular" identifiers. Doing so will result in a syntax error, as demonstrated in the attempted assignment to keyword and below,

... and 5 10
SyntaxError: invalid syntax
The keywords in Python are listed in Figure 2-12. To display the keywords, type help() in the Python shell, and then type keywords (type 'q' to quit).

```
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
```

FIGURE 2-12 Keywords in Python

There are other predefined identifiers that *can* be used as regular identifiers, but should not be. This includes float, int, print, exit, and quit, for example. A simple way to check whether a given identifier is a keyword in Python is given below,

... 'exit' in dir(__builtins__) ... 'exit_program' in dir(__builtins__) True False
LET'S TRY IT
From the Python Shell, enter the following and observe the results.
... yield 5 1000 ... print('Hello')
??? ???
... Yield 5 1000 ... print 5 10
??? ... print('Hello')
???
A **keyword** is an identifier that has predefined meaning in a programming language and therefore cannot be used as a "regular" identifier. Doing so will result in a syntax error.
2.2.5 Let's Apply It—"Restaurant Tab Calculation"
The program below calculates a restaurant tab for a couple based on the use of a gift certificate and the items ordered. This program utilizes the following programming features:

➤ variables
➤ keyboard input
➤ built-in format function ➤ type conversion functions

An example execution of the program is given in Figure 2-13.

Program Execution ...

```
This program will calculate a restaurant tab for a couple with a gift
certificate, with a restaurant tax of 8.0 %
Enter the amount of the gift certificate: 200
Enter ordered items for person 1
Appetizer: 5.50
Entree: 21.50
Drinks: 4.25
Dessert: 6.00

Enter ordered items for person 2
Appetizer: 6.25
Entree: 18.50
Drinks: 6.50
Dessert: 5.50

Ordered items: $ 74.00
Restaurant tax: $ 5.92
Tab: $ -120.08
(negative amount indicates unused amount of gift certificate)
```

FIGURE 2-13 Execution of the Restaurant Tab Calculation Program

The program is given in Figure 2-14. **Lines 1–2** contain comment lines describing what the program does. The remaining comment lines provide an outline of the basic program sections. **Line 5**provides the required initialization of variables in the program, with variable tax assigned to 8% (.08). Variable tax is used throughout the program (in **lines 9 , 35 ,** and **39**). Thus, if the restaurant tax needs to be altered, only this line of the program needs to be changed. (Recall that * is used to denote multiplication in Python, introduced in Chapter 1.)

Lines 8–9 display to the user what the program does. The control character \n as the last character of the print function causes a screen line to be skipped before the next line is displayed. The cost of the menu items ordered is obtained from the user in **lines 15–27**.

Lines 30 and **31** total the cost of the orders for each person, assigned to variables amt_person1 and amt_person2. **Lines 34** and **35** compute the tab, including tax (stored in variable tab). Finally, **lines 38–41** display the cost of the ordered items, followed by the added restaurant tax and the amount due after deducting the amount of the gift certificate. The customers owe any remaining amount.

```
# Restaurant Tab Calculation Program
# This program will calculate a restaurant tab with a gift certificate

# initialization
tax = 0.08

# program greeting
print('This program will calculate a restaurant tab for a couple with')
print('a gift certificate, with a restaurant tax of', tax * 100, '%\n')

# get amount of gift certificate
amt_certificate = float(input('Enter amount of the gift certificate: '))

# cost of ordered items
print('Enter ordered items for person 1')

appetizer_per1 = float(input('Appetizier: '))
entree_per1 = float(input('Entree: '))
drinks_per1 = float(input('Drinks: '))
dessert_per1 = float(input('Dessert: '))

print('\nEnter ordered items for person 2')

appetizer_per2 = float(input('Appetizier: '))
entree_per2 = float(input('Entree: '))
drinks_per2 = float(input('Drinks: '))
dessert_per2 = float(input('Dessert: '))

# total items
amt_person1 = appetizer_per1 + entree_per1 + drinks_per1 + dessert_per1
amt_person2 = appetizer_per2 + entree_per2 + drinks_per2 + dessert_per2

# compute tab with tax
items_cost = amt_person1 + amt_person2
tab = items_cost + items_cost * tax

# display amount owe
print('\nOrdered items: $', format(items_cost, '.2f'))
print('Restaurant tax: $', format(items_cost * tax, '.2f'))
print('Tab: $', format(tab - amt_certificate, '.2f'))
print('(negative amount indicates unused amount of gift certificate)')
```

FIGURE 2-14 Restaurant Tab Calculation Program
A negative amount indicates the amount left on the gift certificate. Built-in function format is used to limit the output to two decimal places.
Self-Test Questions

1. Which of the following are valid assignment statements, in which only variable k has already been assigned a value?
(a)n 5k 11 **(b)**n 5n 11 **(c)**n 1k 510 **(d)**n 11 51

2. What is the value of variable num after the following assignment statements are executed? num 5 0

num 5 num 1 1
num 5 num 1 5

3. Do variables num and k reference the same memory location after the following instructions are executed? (YES/NO)
num 5 10
k 5 num
num 5 num 1 1

4. Which of the following are valid identifiers in Python?
(a)errors **(b)**error_count **(c**) error-count
5. Which of the following are keywords in Python?
(a)and **(b)**As **(c)**while **(d)**until **(e)**NOT

6. Which one of the following is correct for reading and storing an integer value from the user?
(a)n 5int_input('Enter: ') **(b)**n 5int(input('Enter: '))

ANSWERS: 1. (a), 2. 6, 3. No, 4. (a,b), 5. (a,c), 6. (b)

2.3 Operators

Now that we have used numeric and string types in Python, we look at operations that may be performed on them.

2.3.1 What Is an Operator?

An **operator** is a symbol that represents an operation that may be performed on one or more *operands*. For example, the 1 symbol represents the operation of addition. An **operand** is a value that a given operator is applied to, such as operands 2 and 3 in the expression 2 1 3. A **unary operator** operates on only one operand, such as the negation operator in 212. A **binary operator** operates on two operands, as with the addition operator. Most operators in programming languages are binary operators. We look at the arithmetic operators in Python next.

An **operator** is a symbol that represents an operation that may be performed on one or more **operands**. Operators that take one operand are called **unary operators**. Operators that take two operands are called **binary operators**.

2.3.2 Arithmetic Operators

Python provides the arithmetic operators given in Figure 2-15.

The 1, 2, * (multiplication) and / (division) arithmetic operators perform the usual operations. Note that the 2 symbol is used both as a unary operator (for negation) and a binary operator (for subtraction).

20 2 5 → 15 (as binary operator) 210 * 2 → 220 (as unary operator)

Arithmetic Operators		Example	Result
-x	negation	-10	-10
x + y	addition	10 + 25	35
x - y	subtraction	10 - 25	-15
x * y	multiplication	10 * 5	50
x / y	division	25 / 10	2.5
x // y	truncating div	25 // 10	2
		25 // 10.0	2.0
x % y	modulus	25 % 10	5
x ** y	exponentiation	10 ** 2	100

FIGURE 2-15 Arithmetic Operators in Python

Python also includes an exponentiation (**) operator. Integer and floating-point values can be used in both the base and the exponent,

2**4 → 16

2.5 ** 4.5 → 61.76323555016366

Python provides two forms of division. **“true” division** is denoted by a single slash, /. Thus, 25 / 10 evaluates to 2.5. **Truncating division** is denoted by a double slash, //, providing a truncated result based on the type of operands applied to. When both operands are integer values, the result is a truncated integer referred to as **integer division**. When as least one of the operands is a float type, the result is a truncated floating point. Thus, 25 // 10 evaluates to 2, while 25.0 // 10 becomes 2.0. This is summarized in Figure 2-16.

	Operands	result type	example	result
/ Division operator	int, int	float	7 / 5	1.4
	int, float	float	7 / 5.0	1.4
	float, float	float	7.0 / 5.0	1.4
// Truncating division operator	int, int	truncated int ("integer division")	7 // 5	1
	int, float	truncated float	7 // 5.0	1.0
	float, float	truncated float	7.0 // 5.0	1.0

FIGURE 2-16 Division Operators in Python

An example of the use of integer division would be to determine the number of dozen doughnuts for a given number of doughnuts. If variable numDoughnuts had a current value of 29, the number of dozen doughnuts would be calculated by,

numDoughnuts // 12 → 29 // 12 → 2

Lastly, the **modulus operator**(%) gives the remainder of the division of its operands, resulting in a cycle of values. This is shown in Figure 2-17.

Modulo 7		Modulo 10		Modulo 100	
0 % 7	**0**	0 % 10	**0**	0 % 100	**0**
1 % 7	**1**	1 % 10	**1**	1 % 100	**1**
2 % 7	**2**	2 % 10	**2**	2 % 100	**2**
3 % 7	**3**	3 % 10	**3**	3 % 100	**3**
4 % 7	**4**	4 % 10	**4**	.	.
5 % 7	**5**	5 % 10	**5**	.	.
6 % 7	**6**	6 % 10	**6**	96 % 100	**96**
7 % 7	0	7 % 10	**7**	97 % 100	**97**
8 % 7	1	8 % 10	**8**	98 % 100	**98**
9 % 7	2	9 % 10	**9**	99 % 100	**99**
10 % 7	3	10 % 10	0	100 % 100	0
11 % 7	4	11 % 10	1	101 % 100	1
12 % 7	5	12 % 10	2	102 % 100	2

FIGURE 2-17 The Modulus Operator

The modulus and truncating (integer) division operators are complements of each other. For example, 29 // 12 gives the number of dozen doughnuts, while 29 % 12 gives the number of leftover doughnuts (5).

LET’S TRY IT From the Python Shell, enter the following and observe the results.

```
... 10 1 35 ... 4 ** 2 ... 45 // 10.0 ??? ??? ???
... 210 1 35 ... 45 / 10 ... 2025 % 10 ??? ??? ???
... 4 * 2 ... 45 // 10 ... 2025 // 10 ??? ??? ???
```

The **division operator**, /, produces “true division” regardless of its operand types. The **truncating division operator**, //, produces either an integer or float truncated result based on the type of operands applied to. The **modulus operator**(%) gives the remainder of the division of its operands.

2.3.3 Let’s Apply It—“Your Place in the Universe”

The following program (Figure 2-18) calculates the approximate number of atoms that the average person contains, and the percentage of the universe that they comprise. This program utilizes the following programming features:

➤ floating-point scientific notation ➤ built-in format function

Program Execution ...

```
This program will determine your place in the universe.
Enter your weight in pounds: 150
You contain approximately 3.30e+28 atoms
Therefore, you comprise 3.30e-51 % of the universe
```

```
1  # Your Place in the Universe Program
2
3  # This program will determine the approximate number of atoms that a
4  # person consists of and the percent of the universe that they comprise.
5
6  # initialization
7  num_atoms_universe = 10e80
8  weight_avg_person = 70  # 70 kg (154 lbs)
9  num_atoms_avg_person = 7e27
10
11 # program greeting
12 print('This program will determine your place in the universe.')
13
14 # prompt for user's weight
15 weight_lbs = int(input('Enter your weight in pounds: '))
16
17 # convert weight to kilograms
18 weight_kg = 2.2 * weight_lbs
19
20 # determine number atoms in person
21 num_atoms = (weight_kg / 70) * num_atoms_avg_person
22 percent_of_universe = (num_atoms / num_atoms_universe) * 100
23
24 # display results
25 print('You contain approximately', format(num_atoms, '.2e'), 'atoms')
26 print('Therefore, you comprise', format(percent_of_universe, '.2e'),
27       '% of the universe')
```

FIGURE 2-18 Your Place in the Universe Program

Lines 1–4 describe the program. Needed variables num_atoms_universe, weight_avg_ person, and num_atoms_avg_person are initialized in **lines 7–9**. The program greeting is on **line 12**. **Line 15** inputs the person's weight. **Line 18** converts the weight to kilograms for use in the calculations on **lines 21–22** which compute the desired results. Finally, **lines 25–27** display the results.

Self-Test Questions

1. Give the results for each of the following.

(a) 22 * 3 **(b)**15 % 4 **(c)**3 ** 2

2. Give the exact results of each of the following division operations.

(a)5 / 4 **(b)**5 // 4 **(c)**5.0 // 4

3. Which of the expressions in question 2 is an example of integer division?
4. Do any two of the expressions in question 2 evaluate to the exact same result? (YES/NO)

5. How many operands are there in the following arithmetic expression?
2 * 24 1 60 2 10
(a)4 **(b)**3 **(c)**7

6. How many binary operators are there in the following arithmetic expression?
210 1 25 / (16 1 12)
(a)2 **(b)**3 **(c)**4

ANSWERS: 1. (a) 26, (b) 3 (c) 9, 2. (a) 1.25 (b) 1 (c) 1.0, 3. (b), 4. no, 5. (a), 6. (b)

2.4 Expressions and Data Types

Now that we have looked at arithmetic operators, we will see how operators and operands can be combined to form expressions. In particular, we will look at how arithmetic expressions are evaluated in Python. We also introduce the notion of a *data type*.

2.4.1 What Is an Expression?

An **expression** is a combination of symbols that evaluates to a value. Expressions, most commonly, consist of a combination of operators and operands,
4 1 (3 * k)

An expression can also consist of a single literal or variable. Thus, 4, 3, and k are each expressions. This expression has two *subexpressions*, 4 and (3 * k). Subexpression (3 * k) itself has two subexpressions, 3 and k.

Expressions that evaluate to a numeric type are called **arithmetic expressions**. A **subexpression** is any expression that is part of a larger expression. Subexpressions may be denoted by the use of parentheses, as shown above. Thus, for the expression 4 1(3 * 2), the two operands of the addition operator are 4 and (3 * 2), and thus the result it equal to 10. If the expression were instead written as (4 1 3) * 2, then it would evaluate to 14.

Since a subexpression is an expression, any subexpression may contain subexpressions of its own, 4 1 (3 * (2 2 1)) → 4 1 (3 * 1) → 4 1 3 → 7

If no parentheses are used, then an expression is evaluated according to the rules of operator precedence in Python, discussed in the next section.
LET’S TRY IT From the Python Shell, enter the following and observe the results. ... (2 1 3) * 4 ... 2 + ((3 * 4) 2 8) ??? ???
... 2 1 (3 * 4) ... 2 1 3 * (4 2 1) ??? ???
An **expression** is a combination of symbols (or single symbol) that evaluates to a value. A **subexpression** is any expression that is part of a larger expression.
2.4.2 Operator Precedence

The way we commonly represent expressions, in which operators appear between their operands, is referred to as **infix notation**. For example, the expression 4 13 is in infix notation since the 1 operator appears between its two operands, 4 and 3. There are other ways of representing expressions called *prefix* and *postfix* notation, in which operators are placed *before* and *after* their operands, respectively.
The expression 4 1(3 * 5) is also in infix notation. It contains two operators, 1 and *.

The parentheses denote that (3 * 5) is a subexpression. Therefore, 4 and (3 * 5) are the operands of the addition operator, and thus the overall expression evaluates to 19. What if the parentheses were omitted, as given below?

4 1 3 * 5 How would this be evaluated? These are two possibilities, 4 1 **3 * 5** → 4 15 → 19
4 1 3 * 5 → 7 * 5 → 35

Some might say that the first version is the correct one by the conventions of mathematics. However, each programming language has its own rules for the order that operators are applied, called **operator precedence**, defined in an **operator precedence table**. This may or may not be the same as in mathematics, although it typically is. In Figure 2-19, we give the operator precedence table for the Python operators discussed so far. (We will discuss the issue of associativity indicated in Figure 2-19 in the next section.)

Operator	Associativity
** (exponentiation)	right-to-left
- (negation)	left-to-right
* (mult), / (div), // (truncating div), % (modulo)	left-to-right
+ (addition), - (subtraction)	left-to-right

FIGURE 2-19 Operator Precedence of Arithmetic Operators in Python
In the table, higher-priority operators are placed above lower-priority ones. Thus, we see that multiplication is performed before addition when no parentheses are included,

4 1 **3 * 5** → 4 1 15 → 19
In our example, therefore, if the addition is to be performed first, parentheses would be needed, (4 1 3) * 5 → 7 * 5 → 35

As another example, consider the expression below. Following Python's rules of operator precedence, the exponentiation operator is applied first, then the truncating division operator, and finally the addition operator,

4 1 2 ** 5 // 10 → 4 1 32 // 10 → 4 1 3 → 7

Operator precedence guarantees a consistent interpretation of expressions. However, it is good programming practice to use parentheses even when not needed if it adds clarity and enhances readability, without overdoing it. Thus, the previous expression would be better written as,

4 1 (2 ** 5) // 10
LET'S TRY IT From the Python Shell, enter the following and observe the results. ... 2 1 3 * 4 ... 2 * 3 // 4 ??? ???
... 2 * 3 1 4 ... 5 1 42 % 10 ??? ???
... 2 * 3 / 4 ... 2 * 2 ** 3 ??? ???
Operator precedence is the relative order that operators are applied in the evaluation of expressions, defined by a given **operator precedence table**.
2.4.3 Operator Associativity

A question that you may have already had is, "What if two operators have the same level of precedence, which one is applied first?" For operators following

the associative law, the order of evaluation doesn’t matter,

(2 1 3) 1 4 →9 2 1 (3 1 4) → 9
In this case, we get the same results regardless of the order that the operators are applied. Division and subtraction, however, do not follow the associative law,

(a) (8 2 4) 2 2 → 4 2 2 → 2 8 2 (4 2 2) → 8 2 2 → 6 (b) (8 / 4) / 2 → 2 / 2 → 1 8 / (4 / 2) → 8 / 2 → 4 (c) 2 ** (3 ** 2) → 512 (2 ** 3) ** 2 → 64

Here, the order of evaluation does matter. To resolve the ambiguity, each operator has a specified **operator associativity** that defines the order that it and other operators with the same level of precedence are applied (as given in Figure 2-19). All operators in the figure, except for exponentiation, have left-to-right associativity—exponentiation has right-to-left associativity.

LET’S TRY IT From the Python Shell, enter the following and observe the results.

... 6 2 3 1 2 ... 2 * 3 / 4 ... (2 ** 2) ** 3 ??? ??? ??? ... (6 2 3) 1 2 ... 12 % (10 / 2) ... 2 ** (2 ** 3) ??? ??? ??? ... 6 2 (3 1 2) ... 2 ** 2 ** 3
??? ???

Operator associativity is the order that operators are applied when having the same level of precedence, specific to each operator.
2.4.4 What Is a Data Type?

A **data type** is a set of values, and a set of operators that may be applied to those values. For example, the integer data type consists of the set of integers, and operators for addition, subtraction, multiplication, and division, among others. Integers, floats, and strings are part of a set of predefined data types in Python called the **built-in types**.

Data types prevent the programmer from using values inappropriately. For example, it does not make sense to try to divide a string by two, 'Hello' / 2. The programmer knows this by common sense. Python knows it because 'Hello' belongs to the string data type, which does not include the division operation. The need for data types results from the fact that the same internal representation of data can be interpreted in various ways, as shown in Figure 2-20.

FIGURE 2-20 Multiple Interpretations of a Sequence of Bits

The sequence of bits in the figure can be interpreted as a character ('A') or an integer (65). If a programming language did not keep track of the intended type of each value, then the programmer would have to. This would likely lead to undetected programming errors, and would provide even more work for the programmer. We discuss this further in the following section.

Finally, there are two approaches to data typing in programming languages. In **static typing**, a variable is declared as a certain type before it is used, and can only be assigned values of that type. Python, however, uses *dynamic typing*. In **dynamic typing**, the data type of a variable depends only on the type of value that the variable is currently holding. Thus, the same variable may be assigned values of different type during the execution of a program.

A **data type** is a set of values, and a set of operators that may be applied to those values.

2.4.5 Mixed-Type Expressions

A **mixed-type expression** is an expression containing operands of different type. The CPU can only perform operations on values with the same internal representation scheme, and thus only on operands of the same type. Operands of mixed-type expressions therefore must be converted to a common type. Values can be converted in one of two ways—by implicit (automatic) conversion, called *coercion*, or by explicit *type conversion*. We look at each of these next.

A **mixed-type expression** is an expression with operands of different type.

Coercion vs. Type Conversion

Coercion is the *implicit* (automatic) conversion of operands to a common type. Coercion is automatically performed on mixed-type expressions only if the operands can be safely converted, that is, if no loss of information will result. The conversion of integer 2 to floating-point 2.0 below is a safe conversion—the

conversion of 4.5 to integer 4 is not, since the decimal digit would be lost,

2 1 4.5 → **2.0** 1 4.5 → 6.5 safe (automatic conversion of int to float)

Type conversion is the *explicit* conversion of operands to a specific type. Type conversion can be applied even if loss of information results. Python provides built-in **type conversion functions** int() and float(), with the int() function truncating results as given in Figure 2-21.

float(2) 1 4.5 → 2.0 1 4.5 → 6.5 2 1 int(4.5) → 2 1 4 → 6

Conversion Function		Converted Result		Conversion Function		Converted Result
int()	int(10.8)	10		**float()**	float(10)	10.0
	int('10')	10			float('10')	10.0
	int('10.8')	ERROR			float('10.8')	10.8

FIGURE 2-21 Conversion Functions int() and float() in Python
Note that numeric strings can also be converted to a numeric type. In fact, we have already been doing this when using int or float with the input function, num_credits 5 int(input('How many credits do you have? '))
Coercion is the *implicit* (automatic) conversion of operands to a common type.
Type conversion is the *explicit* conversion of operands to a specific type.
2.4.6 Let's Apply It—"Temperature Conversion Program"

The following Python program (Figure 2-22) requests from the user a temperature in degrees Fahrenheit, and displays the equivalent temperature in degrees Celsius. This program utilizes the following programming features:

➤ arithmetic expressions ➤ operator associativity ➤ format function

Program Execution ...

```
This program will convert degrees Fahrenheit to degrees Celsius
Enter degrees Fahrenheit: 100
100.0 degrees Fahrenheit equals 37.8 degrees Celsius
```

```
1  # Temperature Conversion Program (Fahrenheit to Celsius)
2
3  # This program will convert a temperature entered in Fahrenheit
4  # to the equivalent degrees in Celsius
5
6  # program greeting
7  print('This program will convert degrees Fahrenheit to degrees Celsius')
8
9  #get temperature in Fahrenheit
10 fahren = float(input('Enter degrees Fahrenheit: '))
11
12 # calc degrees Celsius
13 celsius = (fahren - 32) * 5 / 9
14
15 # output degrees Celsius
16 print(fahrenheit, 'degrees Fahrenheit equals',
17       format(celsius, '.1f'), 'degrees Celsius')
```

FIGURE 2-22 Temperature Conversion Program

Lines 1–4 contain the program description. **Line 7** provides the program greeting. **Line 10**reads the Fahrenheit temperature entered, assigned to variable fahren. Either an integer or a floatingpoint value may be entered, since the input is converted to float type. **Line 13** performs the calculation for converting Fahrenheit to Celsius. Recall that the division and multiplication operators have the same level of precedence. Since these operators associate left-to-right, the multiplication operator is applied first. Because of the use of the “true” division operator /, the result of the expression will have floating-point accuracy. Finally, **lines 16–17** output the converted temperature in degrees Celsius.

Self-Test Questions

1. What value does the following expression evaluate to?
2 1 9 * ((3 * 12) – 8) / 10
(a)27 **(b)**27.2 **(c)**30.8

2. Evaluate the following arithmetic expressions using the rules of operator precedence in Python.

(a)3 1 2 * 10 **(b)**2 1 5 * 4 1 3 **(c)**20 // 2 * 5 **(d)**2 * 3 ** 2

3. Evaluate the following arithmetic expressions based on Python's rules of operator associativity.
(a)24 // 4 // 2 **(b)**2 ** 2 ** 3

4. Which of the following is a mixed-type expression?
(a)2 1 3.0 **(b)**2 1 3 * 4
5. Which of the following would involve coercion when evaluated in Python?
(a)4.0 1 3 **(b)**3.2 * 4.0
6. Which of the following expressions use explicit type conversion? **(a)**4.0 1 float(3) **(b)**3.2 * 4.0 **(c)**3.2 1 int(4.0)
ANSWERS: 1. (b), 2. (a) 23 (b) 25 (c) 50 (d) 18, 3. (a) 3 (b) 256, 4. (a), 5. (a), 6. (a, c)

COMPUTATIONAL PROBLEM SOLVING 2.5 Age in Seconds Program

We look at the problem of calculating an individual's age in seconds. It is not feasible to determine a given person's age to the exact second. This would require knowing, to the second, when they were born. It would also involve knowing the time zone they were born in, issues of daylight savings time, consideration of leap years, and so forth. Therefore, the problem is to determine an *approximation* of age in seconds. The program will be tested against calculations of age from online resources.

2.5.1 The Problem

The problem is to determine the approximate age of an individual in seconds within 99% accuracy of results from online resources. The program must work for dates of birth from January 1, 1900 to the present.

2.5.2 Problem Analysis

The fundamental computational issue for this problem is the development of an algorithm incorporating approximations for information that is impractical to utilize (time of birth to the second, daylight savings time, etc.), while producing a result that meets the required degree of accuracy.

2.5.3 Program Design

Meeting the Program Requirements

There is no requirement for the form in which the date of birth is to be entered.

We will therefore design the program to input the date of birth as integer values. Also, the program will not perform input error checking, since we have not yet covered the programming concepts for this.

Data Description

The program needs to represent two dates, the user's date of birth, and the current date. Since each part of the date must be able to be operated on arithmetically, dates will be represented by three integers. For example, May 15, 1992 would be represented as follows:

year 51992 month 55 day 5 15
Hard-coded values will also be utilized for the number of seconds in a minute, number of minutes in an hour, number of hours in a day, and the number of days in a year.
Algorithmic Approach

The Python Standard Library module datetime will be used to obtain the current date. (See the Python 3 Programmers' Reference.) We consider how the calculations can be approximated without greatly affecting the accuracy of the results.

We start with the issue of leap years. Since there is a leap year once every four years (with some exceptions), we calculate the average number of seconds in a year over a four-year period that includes a leap year. Since non-leap years have 365 days, and leap years have 366, we need to compute,

numsecs_day 5 (hours per day) * (mins per hour) * (secs per minute)
numsecs_year 5 (days per year) * numsecs_day
avg_numsecs_year 5 (4 * numsecs_year) 1 numsecs_day) // 4
avg_numsecs_month 5 avgnumsecs_year // 12

(one extra day for leap year)

To calculate someone's age in seconds, we use January 1, 1900 as a basis. Thus, we compute two values—the number of seconds from January 1 1900 to the given date of birth, and the number of seconds from January 1 1900 to the current date. Subtracting the former from the latter gives the approximate age,

1900 date of birth current date

num secs 1900 to present
num secs 1900 to date of birth**age in seconds**

Note that if we directly determined the number of seconds between the date of birth and current date, the months and days of each would need to be compared to see how many full months and years there were between the two. Using 1900 as a basis avoids these comparisons. Thus, the rest of our algorithm is given below.

numsecs_1900_to_dob 5 (year_birth 2 1900) * avg_numsecs_year 1
(month_birth 2 1) * avg_numsecs_month 1 (day_birth * numsecs_day)

numsecs_1900_to_today 5 (current_year 2 1900) * avg_numsecs_year 1
(current_month 2 1) * avg_numsecs_month 1
(current_day * numsecs_day)

age_in_secs 5 num_secs_1900_to_today 2 numsecs_1900_to_dob
Overall Program Steps
The overall steps in this program design are in Figure 2-23.

FIGURE 2-23 Overall Steps of the Age in Seconds Program

2.5.4 Program Implementation and Testing
Stage 1—Getting the Date of Birth and Current Date

First, we decide on the variables needed for the program. For date of birth, we use variables month_ birth, day_birth, and year_birth. Similarly, for the current date we use variables current_ month, current_day, and current_year. The first stage of the program assigns each of these values, shown in Figure 2-24.

```
# Age in Seconds Program (Stage 1)
# This program will calculate a person's approximate age in seconds

import datetime

# Get month, day, year of birth
month_birth = int(input('Enter month born (1-12): '))
day_birth = int(input ('Enter day born (1-31): '))
year_birth = int(input('Enter year born (4-digit): '))

# Get current month, day, year
current_month = datetime.date.today().month
current_day = datetime.date.today().day
current_year = datetime.date.today().year

# Test output
print('\nThe date of birth read is: ', month_birth, day_birth,
         year_birth)

print('The current date read is: ', current_month, current_day,
        current_year)
```

FIGURE 2-24 First Stage of Age in Seconds Program

Stage 1 Testing

We add test statements that display the values of the assigned variables. This is to ensure that the dates we are starting with are correct; otherwise, the results will certainly not be correct. The test run below indicates that the input is being correctly read.

Enter month born (1-12): 4
Enter day born (1-31): 12
Enter year born (4-digit): 1981 The date of birth read is: 4 12 1981 The current date read is: 1 5 2010 ...

Stage 2—Approximating the Number of Seconds in a Year/Month/Day

Next we determine the approximate number of seconds in a given year and month, and the exact number of seconds in a day stored in variables avg_numsecs_year, avg_numsecs_month, and numsecs_day, respectively, shown in Figure 2-25.

```
1  # Age in Seconds Program (Stage 2)
2  # This program will calculate a person's approximate age in seconds
3
4  import datetime
5
6  ### Get month, day, year of birth
7  ##month_birth = int(input('Enter month born (1-12): '))
8  ##day_birth = int(input ('Enter day born (1-31): '))
9  ##year_birth = int(input('Enter year born (4-digit): '))
10
11 ### Get current month, day, year
12 ##current_month = datetime.date.today().month
13 ##current_day = datetime.date.today().day
14 ##current_year = datetime.date.today().year
15
16 # Determine number of seconds in a day, average month, and average year
17 numsecs_day = 24 * 60 * 60
18 numsecs_year = 365 * numsecs_day
19
20 avg_numsecs_year = ((4 * numsecs_year) + numsecs_day) // 4
21 avg_numsecs_month = avg_numsecs_year // 12
22
23 # Test output
24 print('numsecs_day ', numsecs_day)
25 print('avg_numsecs_month = ', avg_numsecs_month)
26 print('avg_numsecs_year = ', avg_numsecs_year)
```

FIGURE 2-25 Second Stage of Age in Seconds Program

The lines of code prompting for input are commented out (**lines 6–9** and **11–14**). Since it is easy to comment out (and uncomment) blocks of code in IDLE, we do so; the input values are irrelevant to this part of the program testing.

Stage 2 Testing

Following is the output of this test run. Checking online sources, we find that the number of seconds in a regular year is 31,536,000 and in a leap year is 31,622,400. Thus, our approximation of 31,557,600 as the average number of seconds over four years (including a leap year) is reasonable. The avg_num_seconds_month is directly calculated from variable avg_ numsecs_ year, and numsecs_day is found to be correct.

numsecs_day 86400
avg_numsecs_month 5 2629800
avg_numsecs_year 5 31557600
...

Final Stage—Calculating the Number of Seconds from 1900

Finally, we complete the program by calculating the approximate number of seconds from 1900 to both the current date and the provided date of birth. The difference of these two values gives the approximate age in seconds. The complete program is shown in Figure 2-26.

```
# Age in Seconds Program
# This program will calculate a person's approximate age in seconds

import datetime

# Program greeting
print('This program computes the approximate age in seconds of an')
print('individual based on a provided date of birth. Only dates of')
print('birth from 1900 and after can be computed\n')

# Get month, day, year of birth
month_birth = int(input('Enter month born (1-12): '))
day_birth = int(input ('Enter day born (1-31): '))
year_birth = int(input('Enter year born (4-digit): '))

# Get month, day, year of birth
current_month = datetime.date.today().month
current_day = datetime.date.today().day
current_year = datetime.date.today().year

# Determine number of seconds in a day, average month, and average year
numsecs_day = 24 * 60 * 60
numsecs_year = 365 * numsecs_day

avg_numsecs_year = ((4 * numsecs_year) + numsecs_day) // 4
avg_numsecs_month = avg_numsecs_year // 12

# Calculate approximate age in seconds
numsecs_1900_dob = (year_birth - 1900 * avg_numsecs_year) + \
                    (month_birth - 1 * avg_numsecs_month) + \
                    (day_birth * numsecs_day)

numsecs_1900_today = (current_year - 1900 * avg_numsecs_year) + \
                    (current_month - 1 * avg_numsecs_month) + \
                    (current_day * numsecs_day)

age_in_secs = numsecs_1900_today - numsecs_1900_dob

# output results
print('\nYou are approximately', age_in_secs, 'seconds old')
```

FIGURE 2-26 Final Stage of Age in Seconds Program

We develop a set of test cases for this program. We follow the testing strategy of including “average” as well as “extreme” or “special case” test cases in the test plan. The test results are given in Figure 2-27.

Date of Birth	Expected Results	Actual Results	Inaccuracy	Evaluation
January 1, 1900	3,520,023,010 ± 86,400	1,468,917	99.96 %	failed
April 12, 1981	955,156,351 ± 86,400	518,433	99.94 %	failed
January 4, 2000	364,090,570 ± 86,400	1,209,617	99.64 %	failed
December 31, 2009	48,821,332 ± 86,400	-1,123,203	102.12 %	failed
(day before current date)	86,400 ± 86,400	86,400	0 %	passed

FIGURE 2-27 Results of First Execution of Test Plan

The “correct” age in seconds for each was obtained from an online source. January 1, 1900 was included in the test plan since it is the earliest date (“extreme case”) that the program is required to work for. April 12, 1981 was included as an average case in the 1900s, and January 4, 2000 as an average case in the 2000s. December 31, 2009 was included since it is the last day of the last month of the year. Finally, a test case for a birthday on the day before the current date was included as a special case. (See sample program execution in Figure 2-28). Since these values are continuously changing by the second, we consider any result within one day’s worth of seconds (± 84,000) to be an exact result.

```
This program computes the approximate age in seconds of an
individual based on a provided date of birth. Only ages for
dates of birth from 1900 and after can be computed

Enter month born (1-12): 4
Enter day born (1-31): 12
Enter year born: (4-digit)1981

You are approximately 518433 seconds old
>>>
```

FIGURE 2-28 Example Output of Final Stage Testing

The program results are obviously incorrect, since the result is approximately equal to the average number of seconds in a month (determined above). The only correct result is for the day before the current date. The inaccuracy of each result was calculated as follows for April 12, 1981,

((abs(expected_results – actual_results) – 86,400) / expected_results) * 100 5 ((917,110,352 – 518,433) 2 86400) / 917,110,352) * 100 5 99.93 %

Either our algorithmic approach is flawed, or it is not correctly implemented. Since we didn't find any errors in the development of the first and second stages of the program, the problem must be in the calculation of the approximate age in **lines 29–37**. These lines define three variables: numsecs_1900_dob, numsecs_1900_today, and age_in_secs. We can inspect the values of these variables after execution of the program to see if anything irregular pops out at us.

This program computes the approximate age in seconds of an individual based on a provided date of birth. Only ages for dates of birth from 1900 and after can be computed
Enter month born (1-12): 4
Enter day born (1-31): 12
Enter year born: (4-digit)1981
You are approximately 604833 seconds old
...
... numsecs_1900_dob
2**59961031015**
... numsecs_1900_today
2**59960426182**
...

Clearly, this is where the problem is, since we are getting negative values for the times between 1900 and date of birth, and from 1900 to today. We "work backwards" and consider how the expressions could give negative results. This would be explained if, for some reason, the second operand of the subtraction were greater than the first. That would happen if the expression were evaluated, for example, as

numsecs_1900_dob 5 (year_birth 2 **(1900 * avg_numsecs_year)**) 1 \ (month_birth 2 **(1 * avg_numsecs_month)**) 1 \ (day_birth * numsecs_day)

rather than the following intended means of evaluation,

numsecs_1900_dob 5 (**(year_birth 2 1900)** * avg_numsecs_year) 1 \ (**(month_birth 2 1)** * avg_numsecs_month) 1 \ (day_birth * numsecs_day)

Now we realize! Because we did not use parentheses to explicitly indicate the proper order of operators, by the rules of operator precedence Python evaluated the expression as the first way above, not the second as it should be. This would also explain why the program gave the correct result for a date of birth one day before the current date. Once we make the corrections and re-run the test plan, we get the following results shown in Figure 2-29.

Date of Birth	Expected Results	Actual Results	Inaccuracy
January 1, 1900	3,520,023,010 ± 86,400	3,520,227,600	< .004 %
April 12, 1981	955,156,351 ± 86,400	955,222,200	0 %
January 4, 2000	364,090,570 ± 86,400	364,208,400	< .009 %
December 31, 2009	48,821,332 ± 86,400	48,929,400	< .05 %
(day before current date)	86,400 ± 86,400	86,400	0 %

FIGURE 2-29 Results of Second Execution of Test Plan

These results demonstrate that our approximation of the number of seconds in a year was sufficient to get very good results, well within the 99% degree of accuracy required for this program. We would expect more recent dates of birth to give less accurate results given that there is less time that is approximated. Still, for test case December 31, 2009 the inaccuracy is less than .05 percent. Therefore, we were able to develop a program that gave very accurate results without involving all the program logic that would be needed to consider all the details required to give an exact result.

CHAPTER SUMMARY General Topics

Numeric and String Literals
Limitations of Floating-Point Representation Arithmetic Overflow and Underflow
Character Representation Schemes

(Unicode/ASCII)
Control Characters

String Formatting Implicit and Explicit Line

Joining/Variables and Variable Use/
Keyboard Input/Identifier Naming/ Keywords Arithmetic
Operators/Expressions/Infix Notation Operator Precedence and Associativity
Data Types/Static vs. Dynamic Typing
Mixed-Type Expressions/Coercion and
Type Conversion

Python-Specific Programming Topics

Numeric Literal and String Literal Values in Python Built-in format Function in Python
Variable Assignment and Storage in Python Immutable Values in Python
Identifier Naming and Keywords in Python Arithmetic Operators in Python
Operator Precedence and Associativity in Python Built-in int() and float() Type Conversion

Functions in Python

CHAPTER EXERCISES Section 2.1
1. Based on the information in Figure 2-1, how many novels can be stored in one terabyte of storage?

2. Give the following values in the exponential notation of Python, such that there is only one significant digit to the left of the decimal point.
(a)4580.5034 **(b)**0.00000046004 **(c)** 5000402.000000000006

3. Which of the floating-point values in question 2 would exceed the representation of the precision of floating points typically supported in Python, as mentioned in the chapter?
4. Regarding the built-in format function in Python,
(a) Use the format function to display the floating-point value in a variable named result with three decimal digits of precision.
(b) Give a modified version of the format function in (a) so that commas are included in the displayed results.
5. Give the string of binary digits that represents, in ASCII code,
(a) The string 'Hi!'
(b) The literal string 'I am 24'
6. Give a call to print that is provided one string that displays the following

address on three separate lines. John Doe
123 Main Street
Anytown, Maryland 21009

7. Use the print function in Python to output It's raining today.
Chapter Exercises 75
Section 2.2

8. Regarding variable assignment,
(a) What is the value of variables num1 and num2 after the following instructions are executed? num 5 0
k 5 5
num1 5 num 1 k * 2
num2 5 num 1 k * 2

(b) Are the values id(num1) and id(num2) equal after the last statement is executed? **9.** Regarding the input function in Python,

(a) Give an instruction that prompts the user for their last name and stores it in a variable named last_ name.
(b) Give an instruction that prompts the user for their age and stores it as an integer value named age.
(c) Give an instruction that prompts the user for their temperature and stores it as a float named current_temperature.

10. Regarding keywords and other predefined identifiers in Python, give the result for each of the following, **(a)** 'int' in dir(__builtins__)
(b)'import' in dir(__builtins__)

Section 2.3
11. Which of the following operator symbols can be used as both a unary operator and a binary operator?

1 , 2, *, /
12. What is the exact result of each of the following when evaluated?
(a) 12 / 6.0
(b)21 // 10
(c) 25 // 10.0
13. If variable n contains an initial value of 1, what is the largest value that will be assigned to n after the

following assignment statement is executed an arbitrary number of times?
n 5 (n 1 1) % 100
14. Which of the following arithmetic expressions could potentially result in arithmetic overflow, where n
and k are each assigned integer values?
(a) n * k **(b)** n ** k **(c)** n / k **(d)** n 1 k

Section 2.4

15. Evaluate the following expressions in Python.
(a) 10 2 (5 * 4)
(b)40 % 6
(c) 2 (10 / 3) 1 2

16. Give all the possible evaluated results for the following arithmetic expression (assuming no rules of operator precedence).

2 * 4 1 25 – 5
17. Parenthesize all of the subexpressions in the following expressions following operator precedence in Python.
(a) var1 * 8 2 var2 1 32 / var3
(b)var1 2 6 ** 4 * var2 ** 3

18. Evaluate each of the expressions in question 17 above for var1 5 10, var2 5 30, and var3 5 2. **19.** For each of the following expressions, indicate where operator associativity of Python is used to resolve ambiguity in the evaluation of each expression.
(a) var1 * var2 * var3 2 var4
(b)var1 * var2 / var3
(c) var1 ** var2 ** var3

20. Using the built-in type conversion function float(), alter the following arithmetic expressions so that each is evaluated using floating-point accuracy. Assume that var1, var2, and var3 are assigned integer values. Use the minimum number of calls to function float() needed to produce the results. **(a)** var1 1 var2 * var3
(b)var1 // var2 1 var3
(c) var1 // var2 / var3

PYTHON PROGRAMMING EXERCISES

P1. Write a Python program that prompts the user for two integer values and displays the result of the first number divided by the second, with exactly two decimal places displayed.
P2. Write a Python program that prompts the user for two floating-point values and displays the result of the first number divided by the second, with exactly six decimal places displayed.
P3. Write a Python program that prompts the user for two floating-point values and displays the result of the first number divided by the second, with exactly six decimal places displayed in scientific notation.
P4. Write a Python program that prompts the user to enter an upper or lower case letter and displays the corresponding Unicode encoding.
P5. Write a Python program that allows the user to enter two integer values, and displays the results when each of the following arithmetic operators are applied. For example, if the user enters the values 7 and 5, the output would be,
7 1 5 5 12
7 2 5 5 2
7 * 5 5 35
7 / 5 5 1.40
7 // 5 5 1
7 % 5 5 2
7 ** 5 5 16,807

All floating-point results should be displayed with two decimal places of accuracy. In addition, all values should be displayed with commas where appropriate.
PROGRAM MODIFICATION PROBLEMS

M1. Modify the Restaurant Tab Calculation program of section 2.2.5 so that, instead of the restaurant tax being hard coded in the program, the tax rate is entered by the user.
M2. Modify the Restaurant Tab Calculation program of section 2.2.5 so that, in addition to displaying the total of the items ordered, it also displays the total amount spent on drinks and dessert, as well as the percentage of the total cost of the meal (before tax) that these items comprise. Display the monetary amount rounded to two decimal places.
M3. Modify the Your Place in the Universe program in section 2.3.3 for international users, so that the user enters their weight in kilograms, and not in pounds.
M4. Modify the Temperature Conversion program in section 2.4.6 to convert

from Celsius to Fahrenheit instead. The formula for the conversion is f 5 (c * 9/5) 1 32.

Program Development Problems 77

M5. Modify the Age in Seconds program so that it displays the estimated age in number of days, hours, and minutes.
M6. Modify the Age in Seconds program so that it determines the difference in age in seconds of two friends.

PROGRAM DEVELOPMENT PROBLEMS

D1. Losing Your Head over Chess
The game of chess is generally believed to have been invented in India in the sixth century for a ruling king by one of his subjects. The king was supposedly very delighted with the game and asked the subject what he wanted in return. The subject, being clever, asked for one grain of wheat on the first square, two grains of wheat on the second square, four grains of wheat on the third square, and so forth, doubling the amount on each next square. The king thought that this was a modest reward for such an invention. However, the total amount of wheat would have been more than 1,000 times the current world production.

Develop and test a Python program that calculates how much wheat this would be in pounds, using the fact that a grain of wheat weighs approximately 1/7,000 of a pound.

D2. All That Talking
Develop and test a Python program that determines how much time it would take to download all instances of every word ever spoken. Assume the size of this information as given in Figure 2-1. The download speed is to be entered by the user in million of bits per second (mbps). To find your actual connection speed, go to the following website (from Intel Corporation) or similar site,

www.intel.com/content/www/us/en/gamers/broadband-speed-test.html

Because connection speeds can vary, run this connection speed test three times. Take the average of three results, and use that as the connection speed to enter into your program. Finally, determine what is an appropriate unit of time to express your program results in: minutes? hours? days? other?

D3. Pictures on the Go

Develop and test a Python program that determines how many images can be stored on a given size USB (flash) drive. The size of the USB drive is to be entered by the user in gigabytes (GB). The number of images that can be stored must be calculated for GIF, JPEG, PNG, and TIFF image file formats. The program output should be formatted as given below.

Enter USB size (GB): 4

xxxxx images in GIF format can be stored xxxxx images in JPEG format can be stored xxxxx images in PNG format can be stored xxxxx images in TIFF format can be stored

The ultimate file size of a given image depends not only on the image format used, but also on the image itself. In addition, formats such as JPEG allow the user to select the degree of compression for the image quality desired. For this program, we assume the image compression ratios given below. Also assume that all the images have a resolution of 800 3 600 pixels.

Thus, for example, a 800 3 600 resolution image with 16-bit (2 bytes) color depth would have a total number of bytes of 800 3 600 3 2 5 960,000. For a compression rate of 25:1, the total number of bytes needed to store the image would be 960000/25 5 38400.

Finally, assume that a GB (gigabyte) equals 1,000,000,000 bytes, as given in Figure 2.1.

Format	Full Name	Color Depth		Compression	
GIF	Graphics Interchange Format	256 colors	8 bits	lossless	5:1
JPEG	Joint Photographic Experts Group	16 million colors	24 bits	lossy	25:1
PNG	Portable Network Graphics	16 million colors	24 bits	lossless	8:1
TIFF	Tagged Image File Format	280 trillion colors	48 bits	lossless	n/a

Note that a “lossless” compression is one in which no information is lost. A “lossy” compression does lose some of the original information.

D4. Life Signs
Develop and test a program that prompts the user for their age and determines approximately how many breaths and how many heartbeats the person has had in their life. The average respiration (breath) rate of people changes during different stages of development. Use the breath rates given below for use in your program:

Breaths per Minute
Infant 30–60 1–4 years 20–30
5–14 years 15–25
adults 12–20

For heart rate, use an average of 67.5 beats per second.

Control Structures CHAPTER 3

In Chapter 2 we looked at the “nuts and bolts” of programming. In this chapter, we discuss the three fundamental means of controlling the order of execution of instructions within a program, referred to as sequential, selection, and iterative control.

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Explain what a control structure is
♦ Explain the difference between sequential, selection, and iterative control ♦ Describe and use Boolean operators
♦ Explain the notion of logically equivalent Boolean expressions ♦ Explain what is meant by an infinite loop
♦ Explain the difference between a definite and indefinite loop ♦ Explain the use of indentation in Python
♦ Effectively use if statements in Python for selection control ♦ Effectively implement multi-way selection in Python
♦ Effectively use while statements in Python for iterative control

CHAPTER CONTENTS
Motivation

Fundamental Concepts
3.1 What Is a Control Structure? 3.2 Boolean Expressions (Conditionals) 3.3 Selection Control
3.4 Iterative Control
Computational Problem Solving 3.5 Calendar Month Program

MOTIVATION

The first electronic computers over sixty years ago were referred to as “Electronic Brains.” This gave the misleading impression that computers could “think.” Although very complex in their design, computers are machines that simply do, step-by-step (instruction-by-instruction), what they are told. Thus, there is no more intelligence in a computer than what it is instructed to do.

What computers can do, however, is to exe cute a series of instructions very quickly and very

reliably. It is the speed in which instructions can be executed that gives computers their power (see Figure 3-1), since the execution of many simple instructions can result in very complex behavior. And thus this is the enticement of computing. A computer can accomplish any task for which there is an algorithm for doing so. The instructions could be for something as simple as sorting lists, or as ambitious as

performing intelligent tasks that as of now only humans are capable of performing.

In this chapter, we look at how to control the order that instructions are executed in Python.

Term	Number of Floating Point Operations / Second		Device
Megaflops	10^6	1 million FLOPS	Supercomputers (1970s)
Gigaflops	10^9	1 billion FLOPS	CPU (single core)
Teraflops	10^{12}	1 trillion FLOPS	CPU (multi-core)
Petaflops	10^{15}	1 quadrillion FLOPS	Supercomputers (current)
Exaflops	10^{18}	1 quintillion FLOPS	Supercomputers in 2020 (projected)

FIGURE 3-1 Processing Speed—Floating-Point Operations per Second (FLOPS)

FUNDAMENTAL CONCEPTS

3.1 What Is a Control Structure?

C *ontrol flow* is the order that instructions are executed in a program. A **control statement** is a statement that determines the control flow of a set of instructions. There are three fundamental forms of control that programming languages provide— *sequential control*, *selection control*, and *iterative control.*

Sequential control is an implicit form of control in which instructions are executed in the order that they are written. A program consisting of only sequential control is referred to as a “straight-line program.” The program examples in Chapter 2 are all straight-line programs. **Selection control** is provided by a control statement that *selectively executes* instructions, while **iterative control** is provided by an iterative control statement that *repeatedly executes* instructions. Each is based on a given condition. Collectively a set of instructions and the control statements controlling their execution is called a **control structure**.

Few programs are straight-line programs. Most use all three forms of control, depicted in Figure 3-2. We look at selection control and iterative control next.

FIGURE 3-2 if Statement

A **control statement** is a statement that determines the control flow of a set of instructions. A **control structure** is a set of instructions and the control statements controlling their execution. Three fundamental forms of control in programming are **sequential**, **selection**, and **iterative control**.

3.2 Boolean Expressions (Conditions)

The **Boolean data type** contains two Boolean values, denoted as **True** and **False** in Python. A **Boolean expression** is an expression that evaluates to a Boolean value. Boolean expressions are used to denote the conditions for selection and iterative control statements. We look at the use of Boolean expressions next.

The **Boolean data type** contains two Boolean values, denoted as **True** and **False** in Python. A **Boolean expression** is an expression that evaluates to a Boolean value.
3.2.1 Relational Operators

The relational operators in Python perform the usual comparison operations, shown in Figure 3-3. Relational expressions are a type of Boolean expression, since they evaluate to a Boolean result. These operators not only apply to numeric values, but to any set of values that has an ordering, such as strings.

Note the use of the **comparison operator**, 55, for determining if two values are equal. This, rather than the (single) equal sign, 5, is used since the equal sign is used as the assignment operator. This is often a source of confusion for new programmers,

num 510 variable num is assigned the value 10
num 55 10 variable num is compared to the value 10

Relational Operators	Example	Result
== equal	10 == 10	True
!= not equal	10 != 10	False
< less than	10 < 20	True
> greater than	'Alan' > 'Brenda'	False
<= less than or equal to	10 <= 10	True
>= greater than or equal to	'A' >= 'D'	False

FIGURE 3-3 The Relational Operators

Also, ! 5 is used for inequality simply because there is no keyboard character for the fi symbol. String values are ordered based on their character encoding, which normally follows a
lexographical (**dictionary**) **ordering**. For example, 'Alan' is less than 'Brenda' since the
Unicode (ASCII) value for 'A' is 65, and 'B' is 66. However, 'alan' is greater than (comes
after) 'Brenda' since the Unicode encoding of lowercase letters (97, 98, . . .) comes *after* the
encoding of uppercase letters (65, 66, . . .). Recall from Chapter 2 that the encoding of any character can be obtained by use of the ord function.

LET’S TRY IT From the Python Shell, enter the following and observe the results.

... 10 55 20 ... '2' , '9' ... 'Hello' 55 "Hello" ??? ??? ???
... 10 !5 20 ... '12' , '9' ... 'Hello' , 'Zebra' ??? ??? ???
... 10 ,5 20 ... '12' . '9' ... 'hello' , 'ZEBRA' ??? ??? ???

The relational operators 55, !5, ,, ., ,5, .5 can be applied to any set of values that has an ordering.
3.2.2 Membership Operators

Python provides a convenient pair of **membership operators**. These operators can be used to easily determine if a particular value occurs within a specified list of values. The membership operators are given in Figure 3-4.

The in operator is used to determine if a specific value is in a given list, returning True if found, and False otherwise. The not in operator returns the opposite result. The list of values surrounded by matching parentheses in the figure are called *tuples* in Python. Tuples (and lists) are covered in Chapter 4.

Membership Operators	Examples	Result
in	10 in (10, 20, 30)	True
	red in ('red','green','blue')	True
not in	10 not in (10, 20, 30)	False

FIGURE 3-4 The Membership Operators

The membership operators *can also be used to check if a given string occurs within another string*,

... 'Dr.' in 'Dr. Madison'

True

As with the relational operators, the membership operators can be used to construct Boolean expressions.

LET'S TRY IT From the Python Shell, enter the following and observe the results.

... 10 in (40, 20, 10) ... grade 5 'A'

??? ... grade in ('A','B','C','D','F')

???

... 10 not in (40, 20, 10)

??? ... city 5 'Houston'

... city in ('NY', 'Baltimore', 'LA') ... .25 in (.45, .25, .65) ???

???

Python provides membership operators in and not in for determining if a specific value is in (or not in) a given list of values.

3.2.3 Boolean Operators

George Boole, in the mid-1800s, developed what we now call *Boolean algebra*. His goal was to develop an algebra based on true/false rather than numerical

values. Boolean algebra contains a set of **Boolean** (**logical**) **operators**, denoted by and, or, and not in Python. These logical operators can be used to construct arbitrarily complex Boolean expressions. The Boolean operators are shown in Figure 3-5.

Logical and is true only when *both* its operands are true—otherwise, it is false. Logical or is true when *either or both* of its operands are true, and thus false only when both operands are false. Logical not simply reverses truth values—not False equals True, and not True equals False.

x	y		x and y	x or y	not x
False	False		False	False	True
True	False		False	True	False
False	True		False	True	
True	True		True	True	

FIGURE 3-5 Boolean Logic Truth Table

One must be cautious when using Boolean operators. For example, in mathematics, to denote that a value is within a certain range is written as
1 ,5 num ,5 10
In most programming languages, however, this expression does not make sense. To see why, let's assume that num has the value 15. The expression would then be evaluated as follows, 1 ,5 num ,5 10 → 1 ,5 15 ,5 10 → True ,5 10 → **?!?**

It does not make sense to check if True is less than or equal to 10. (Some programming languages would generate a mixed-type expression error for this.) The correct way of denoting the condition is by use of the Boolean and operator,

1 ,5 num and num ,5 10

In some languages (such as Python), Boolean values True and False have integer values 1 and 0, respectively. In such cases, the expression 1 ,5num ,510 would evaluate to True ,510 would evaluate to 1 ,510, which equals True. This would not be the correct result for this expression, however. Let's see what we get when we do evaluate this expression in the Python shell,

. .. num 5 15

... 1 ,5 num ,5 10
False

We actually get the correct result, False. So what is going on here? The answer is that Python is playing a trick here. For Boolean expressions of the particular form,
value1 ,5 *var* ,5 *value2*
Python automatically rewrites this before performing the evaluation,
value1 ,5 *var* and *var* ,5 *value2*
Thus, it is important to note that expressions of this form are handled in a special way in Python, and would not be proper to use in most other programming languages.

One must also be careful in the use of and/or Boolean operators. For example, not(num 55 0 and num 5 5 1) is True for any value of num, as is (num ! 5 0) or (num ! 5 1), and therefore are not useful expressions. The Boolean expression num , 0 and num . 10 is also useless since it is always False.

Finally, Boolean literals True and False are never quoted. Doing so would cause them to be taken as string values ('True'). And as we saw, Boolean expressions do not necessarily contain Boolean operators. For example, 10 5 20 is a Boolean expression. By definition, Boolean literals True and False are Boolean expressions as well.

LET’S TRY IT From the Python Shell, enter the following and observe the results.

... True and False ... (10 , 0) and (10 . 2) ??? ???
... True or False ... (10 , 0) or (10 . 2) ??? ???
... not(True) and False ... not(10 , 0) or (10 . 2) ??? ???
... not(True and False) ... not(10 , 0 or 10 . 2) ??? ???

Boolean operators in Python are denoted by and, or, and not.
3.2.4 Operator Precedence and Boolean Expressions

The operator precedence (and operator associativity) of arithmetic operators was given in Chapter 2. Operator precedence also applies to Boolean operators. Since Boolean expressions can contain arithmetic as well as relational and Boolean operators, the precedence of all operators needs to be collectively applied. An updated operator precedence table is given in Figure 3-6.

Operator	Associativity
** (exponentiation)	right-to-left
- (negation)	left-to-right
* (mult), / (div), // (truncating div), % (modulo)	left-to-right
+ (addition), - (subtraction)	left-to-right
<, >, <=, >=, !=, == (relational operators)	left-to-right
not	left-to-right
and	left-to-right
or	left-to-right

FIGURE 3-6 Operator Precedence of Arithmetic, Relational, and Boolean Operators

As before, in the table, higher-priority operators are placed above lower-priority ones. Thus, we see that *all arithmetic operators are performed before any relational or Boolean operator*, 10 20 1 30 → 30 50 → True

In addition, *all of the relational operators are performed before any Boolean operator*, 10 20 and 30 , 20 → True and False → False

10 20 or 30 , 20 → True or False → True

And as with arithmetic operators, Boolean operators have various levels of precedence. Unary Boolean operator not has higher precedence than and, and Boolean operator and has higher precedence than or.

10 , 20 and 30 , 20 or 30 , 40 → True and False or True
→ False or True → True not 10 , 20 or 30 , 20 → not True or False
→ False or False → False

As with arithmetic expressions, it is good programming practice to use parentheses, even if not needed, to add clarity and enhance readability. Thus, the above expressions would be better written by denoting at least some of the subexpressions,

(10 , 20 and 30 , 20) or (30 , 40)
(not 10 , 20) or (30 , 20)
if not all subexpressions,
((10 , 20) and (30 , 20)) or (30 , 40)
(not (10 , 20)) or (30 , 20)

Finally, note from Figure 3.6 above that all relational and Boolean operators associate from left to right.

LET’S TRY IT
From the Python Shell, enter the following and observe the results.

... not True and False ... 10 , 0 and not 10 . 2 ??? ???
... not True and False or True ... not (10 , 0 or 10 , 20) ??? ???

3.2.5 Short-Circuit (Lazy) Evaluation

There are differences in how Boolean expressions are evaluated in different programming languages. For logical and, if the first operand evaluates to false, then regardless of the value of the second operand, the expression is false. Similarly, for logical or, if the first operand evaluates to true, regardless of the value of the second operand, the expression is true. Because of this, some programming languages do not evaluate the second operand when the result is known by the first operand alone, called **shortcircuit (lazy) evaluation**. Subtle errors can result if the programmer is not aware of this. For example, the expression

if n ! 5 0 and 1/n , tolerance:

would evaluate without error for all values of n when short-circuit evaluation is used. If programming in a language not using short-circuit evaluation, however, a “divide by zero” error would result when n is equal to 0. In such cases, the proper construction would be,

if n ! 5 0:
if 1/n , tolerance:
In the Python programming language, short-circuit evaluation is used.
In **short-circuit (lazy) evaluation**, the second operand of Boolean operators and and or is not evaluated if the value of the Boolean expression can be determined from the first operand alone.
3.2.6 Logically Equivalent Boolean Expressions

In numerical algebra, there are arithmetically equivalent expressions of different form. For example, x(y 1 z) and xy 1 xz are equivalent for any numerical values x, y, and z. Similarly, there are *logically* equivalent **Boolean expressions** of different form. We give some examples in Figure 3-7.

FIGURE 3-7 Logically Equivalent Conditional Expressions

The range of values satisfying each set of expressions is shaded in the figure. Both expressions in (1) are true for any value except 0. The expressions in (2) are true for any value except 0 and 6. The expressions in (3) are only true for values in the range 0 through 6, inclusive. The expressions in (4) are true for all values *except* 0 through 6, inclusive. Figure 3-8 lists common forms of logically equivalent expressions.

Logically Equivalent Boolean Expressions		
x < y	is equivalent to	not(x >= y)
x <= y	is equivalent to	not(x > y)
x == y	is equivalent to	not(x != y)
x != y	is equivalent to	not(x == y)
not(x and y)	is equivalent to	(not x) or (not y)
not(x or y)	is equivalent to	(not x) and (not y)

FIGURE 3-8 Forms of Logically Equivalent Boolean Expressions
The last two equivalences above are referred to as De Morgan's Laws.
LET'S TRY IT
From the Python Shell, enter the following and observe the results.

... 10 , 20 ... not(10 , 20 and 10 , 30) ??? ???
... not(10 .5 20) ... (not 10 , 20) or (not 10 , 30) ??? ???
... 10 !5 20 ... not(10 , 20 or 10 , 30) ??? ???
... not (10 55 20) ... (not 10 , 20) and (not 10 , 30) ??? ???

There are logically equivalent **Boolean expressions** of different form.
Self-Test Questions
1. Three forms of control in programming are sequential, selection, and _____ control.
2. Which of the following expressions evaluate to True?
(a)10 .5 8 **(b)**8 ,5 10 **(c)**10 55 8 **(d)**10 !5 8 **(e)**'8' , '10' **3.** Which of the following Boolean expressions evaluate to True?
(a)'Dave' , 'Ed' **(b)**'dave' , 'Ed' **(c)**'Dave' , 'Dale'

4. What is the value of variable num after the following is executed?
... num 5 10
... num 5 num 1 5
... num 55 20
... num 5 num 1 1

5. What does the following expression evaluate to for name equal to 'Ann'? name in ('Jacob', 'MaryAnn', 'Thomas')
6. Evaluate the following Boolean expressions using the operator precedence

rules of Python. **(a)**10 .5 8 and 5 !5 3 **(b)** 10 .5 8 and 5 55 3 or 14 , 5
7. Which one of the following Boolean expressions is not logically equivalent to the other two? **(a)**not(num , 0 or num . 10)
(b)num . 0 and num , 10
(c)num .5 0 and num ,5 10

ANSWERS: 1. Iterative, 2. (a,b,d), 3. (a), 4. 16, 5. False, 6. (a) True, (b) False, 7. (b)

3.3 Selection Control

A **selection control statement** is a control statement providing selective execution of instructions. A *selection control structure* is a given set of instructions and the selection control statement(s) controlling their execution. We look at the *if statement* providing selection control in Python next.

A **selection control statement** is a control statement providing selective execution of instructions.

3.3.1 If Statement

An **if statement** is a selection control statement based on the value of a given Boolean expression. The if statement in Python is depicted in Figure 3-9.

if statement	Example use	

`if condition:`
`  statements`
`else:`
`  statements`
	`if grade >= 70:`
	`  print('passing grade')`
	`else:`
	`  print('failing grade')`
		`if grade == 100:`
		`  print('perfect score!')`

FIGURE 3-9 if Statement

Note that if statements may omit the “else” part. A version of the temperature conversion program from Chapter 2 using an if statement is given in Figure 3-10.

This program extends the original program by converting Celsius to Fahrenheit, as well as Fahrenheit to Celsius. The if statement (**line 13**) selects the appropriate set of instructions to execute based on user input ('F' for Fahrenheit to Celsius, and 'C' for Celsius to Fahrenheit). A statement that contains other statements, such as the if statement, is called a **compound statement**. We look at Python’s use of indentation in compound statements next.

```
# Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)

# Display program welcome
print('This program will convert temperatures (Fahrenheit/Celsius)')
print('Enter (F) to convert Fahrenheit to Celsius')
print('Enter (C) to convert Celsius to Fahrenheit')

# Get temperature to convert
which = input('Enter selection: ')
temp = int(input('Enter temperature to convert: '))

# Determine temperature conversion needed and display results
if which == 'F':
    converted_temp = (temp - 32) * 5/9
    print(temp, 'degrees Fahrenheit equals', converted_temp, 'degrees Celsius')
else:
    converted_temp = (9/5 * temp) + 32
    print(temp, 'degrees Celsius equals', converted_temp, 'degrees Fahrenheit')
```

FIGURE 3-10 Temperature Conversion (Two-Way Conversion)

An **if statement** is a selection control statement based on the value of a given Boolean expression. Statements that contain other statements are referred to as a **compound statement**.

3.3.2 Indentation in Python

One fairly unique aspect of Python is that the amount of indentation of each program line is significant. In most programming languages, indentation has no affect on program logic—it is simply used to align program lines to aid readability. In Python, however, indentation is used to associate and group statements, as shown in Figure 3-11.

FIGURE 3-11 Compound Statement in Python

A **header** in Python is a specific keyword followed by a colon. In the figure, the

if-else statement contains two headers, “if which 5 5 'F':” containing keyword if, and “else:” consisting only of the keyword else. Headers that are part of the same compound statement must be indented the same amount—otherwise, a syntax error will result.

The set of statements following a header in Python is called a **suite** (commonly called a **block**). The statements of a given suite must all be indented the same amount. A header and its associated suite are together referred to as a **clause**. A compound statement in Python may consist of one or more clauses. While four spaces is commonly used for each level of indentation, any number of spaces may be used, as shown in Figure 3-12.

Valid indentation		Invalid indentation	

(a) `if condition:`
`  statement`
`  statement`
`else:`
`  statement`
`  statement`
	(b) `if condition:`
	`  statement`
	`  statement`
	`else:`
	`  statement`
	`  statement`
		(c) `if condition:`
		`  statement`
		`  statement`
		`else:`
		`  statement`
		`  statement`
			(d) `if condition:`
			`  statement`
			`  statement`
			`else:`
			`  statement`
			`  statement`

FIGURE 3-12 Compound Statements and Indentation in Python

Both (a) and (b) in the figure are properly indented. In (a), both suites have the same amount of indentation. In (b), each suite has a different amount of indentation. This is syntactically correct (although not good practice) since the amount of indentation within each suite is consistent. Both (c) and (d) are examples of invalid indentation, and thus syntactically incorrect. In (c), the if and else headers of the if statement are not indented the same amount. In (d), the headers are indented the same amount. However, the statements within the second suite are not properly aligned. Finally, note that the suite following a header can itself be a compound statement (another if statement, for example). Thus, compound statements may be nested one within another. We look at nested compounded statements next.

LET’S TRY IT

From IDLE, create and run a Python program containing the code on the left and observe the results. Modify and run the code to match the version on the right and again observe the results. Make sure to indent the code exactly as shown.

grade 5 90

if grade .5 70:
print('passing grade')

else:
print('failing grade') grade 5 90
if grade .5 70:

print('passing grade') else:
print('failing grade')

A **header** in Python starts with a keyword and ends with a colon. The group of statements following a header is called a **suite**. A header and its associated suite are together referred to as a **clause**.

3.3.3 Multi-Way Selection

In this section, we look at the two means of constructing multi-way selection in Python—one involving multiple nested if statements, and the other involving a single if statement and the use of elif headers.

Nested if Statements

There are often times when selection among more than two sets of statements (suites) is needed. For such situations, if statements can be nested, resulting in **multi-way selection**. An example of this is given in Figure 3-13.

Nested if statements	Example use

`if condition:`
`  statements`
`else:`
`  if condition:`
`  statements`
`  else:`
`  if condition:`
`  statements`
 	
`  etc.`
	`if grade >= 90:`
	`  print('Grade of A')`
	`else:`
	`  if grade >= 80:`
	`  print('Grade of B')`
	`  else:`
	`  if grade >= 70:`
	`  print('Grade of C')`
	`  else:`
	`  if grade >= 60:`
	`  print('Grade of D')`
	`  else:`
	`  print('Grade of F')`

FIGURE 3-13 Multi-way Selection Using if Statements

The nested if statements on the right result in a 5-way selection. In the first if statement, if variable grade is greater than or equal to 90, then 'Grade of A' is displayed. Therefore, its else suite is not executed, containing the remaining if statements. If grade is less than 90, the else suite is executed. If grade is greater than or equal to 80, 'Grade of B' is displayed and the rest of the if statements in *its* else suite are skipped, and so on. The final else clause is executed only if all the previous conditions fail, displaying 'Grade of F'. This is referred to as a *catch-all* case. As an example use of nested if statements and a check for invalid input in a program, we give a revised version of the temperature conversion program from Figure 3-10 in Figure 3-14.

```
1  # Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)
2
3  # Display program welcome
4  print('This program will convert temperatures (Fahrenheit/Celsius)')
5  print('Enter (F) to convert Fahrenheit to Celsius')
6  print('Enter (C) to convert Celsius to Fahrenheit')
7
8  # Get temperature to convert
9  which = input('Enter selection: ')
10 temp = int(input('Enter temperature to convert: '))
11
12 # Determine temperature conversion needed and display results
13 if which == 'F':
14     converted_temp = format((temp - 32) * 5.0/9.0, '.1f')
15     print(temp, 'degrees Fahrenheit equals', converted_temp, 'degrees Celsius')
16 else:
17     if which == 'C':
18        converted_temp = format((9.0/5.0 * temp) + 32, '.1f')
19        print(temp,'degrees Celsius equals',converted_temp,'degrees Fahrenheit')
20     else:
21         print('INVALID INPUT')
```

FIGURE 3-14 Temperature Conversion Program (Input Error Dectection)

In this version, there is a catch-all clause (**line 20**) for handling invalid input. We next look at a more concise means of denoting multi-way selection in Python.

LET’S TRY IT

From IDLE, create and run a simple program containing the code below and observe the results. Make sure to indent the code exactly as shown.

```
credits 5 45
if credits .5 90:
print('Senior')
else:
if credits 60:
print('Junior')
```

else:
if credits 30:
print('Sophomore')
else:
if credits 1:
print('Freshman')
else:
print('* No Earned Credits *')
If statements can be nested in Python, resulting in **multi-way selection**.
The **elif** Header in Python

If statements may contain only one else header. Thus, if-else statements must be nested to achieve multi-way selection. Python, however, has another header called elif ("else-if") that provides multi-way selection in a *single* if statement, shown in Figure 3-15.

All the headers of an if-elif statement are indented the same amount, thus avoiding the deeply nested levels of indentation with the use of if-else statements. A final else clause may be used for "catch-all" situations. We next look at iterative control in Python.

```
if grade >= 90:
    print('Grade of A')
elif grade >= 80:
    print('Grade of B')
elif grade >= 70:
    print('Grade of C')
elif grade >= 60:
    print('Grade of D')
else:
    print('Grade of F')
```

FIGURE 3-15 The elif Header in Python
LET'S TRY IT From IDLE, create and run a Python program containing the code below and observe the results. Make sure to indent the code exactly as shown.
credits 5 45

if credits .5 90:
print('Senior')

```
elif credits .5 60:
print('Junior')
elif grade .5 30:
print('Sophomore')
elif grade .5 1:
print('Freshman')
else:
print('* No Earned Credits *')
```

If statements may contain any number of **elif headers**, providing for multi-way selection.

3.3.4 Let's Apply It—Number of Days in Month Program

The following Python program (Figure 3-16) prompts the user for a given month (and year for February), and displays how many days are in the month. This program utilizes the following programming features:

➤ if statement ➤ elif header

Lines 1–4 provide the program header and program greeting. On **line 7**, variable valid_input is initialized to True for the input error-checking performed. **Line 10** prompts the user for the month, read as an integer value (1–12), and stores in variable month. On **line 15** the month of February is checked for. February is the only month that may have a different number of days—28 for a regular year, and 29 for leap years. Thus, when February (2) is entered, the user is also prompted for the year (**line 16**). If the year is a leap year, then variable num_days is set to 29— otherwise, it is set to 28.

Generally, if a year is (evenly) divisible by 4, then it is a leap year. However, there are a couple of exceptions. If the year is divisible by 4 but is also divisible by 100, then it is *not* a leap year— unless, it is also divisible by 400, then it is. For example, 1996 and 2000 were leap years, but 1900 was not. This condition is given below.

(year % 4 55 0) and (not (year % 100 55 0) or (year % 400 55 0))
Thus, the conditions for which this Boolean expression is true are,
(year % 4 55 0) and not (year % 100 55 0) and
(year % 4 55 0) and (year % 400 55 0)

Program Execution ...

```
This program will determine the number of days in a given month

Enter the month (1-12): 14
* Invalid Value Entered - 14 '*'
>>>

This program will determine the number of days in a given month

Enter the month (1-12): 2
Please enter the year (e.g., 2010): 2000
There are 29 days in the month
```

```
1  # Number of Days in Month Program
2
3  # program greeting
4  print('This program will display the number of days in a given month\n')
5
6  # init
7  valid_input = True
8
9  # get user input
10 month = int(input('Enter the month (1-12): '))
11
12 # determine num of days in month
13
14 # february
15 if month == 2:
16     year = int(input('Please enter the year (e.g., 2010): '))
17
18     if (year % 4 == 0) and (not (year % 100 == 0) or (year % 400 == 0)):
19         num_days = 29
20     else:
21         num_days = 28
22
23 # january, march, may, july, august, october, december
24 elif month in (1, 3, 5, 7, 8, 10, 12):
25     num days = 31
26
27 # april, june, september, november
28 elif month in (4,6,9,11):
29     num_days = 30
30
31 # invalid input
32 else:
33     print('* Invalid Value Entered - ', month, '*')
34     valid_input = False
35
36 # output result
37 if valid_input:
38     print('There are', num_days, 'days in the month')
```

FIGURE 3-16 Number of Days in Month Program

Line 24 checks if month is equal to 1, 3, 5, 7, 8, 10, or 12. If true, then

num_days is assigned to 31. If not true, **line 28** checks if month is equal to 4, 6, 9, or 11 (all the remaining months except February). If true, then num_days is assigned to 30. If not true, then an invalid month (number) was entered, and valid_input is set to False. Finally, the number of days in the month is displayed only if the input is valid (**line 38**).

Self-T est Questions

1. All if statements must contain either an else or elif header. (TRUE/FALSE)

2. A compound statement is,
(a) A statement that spans more than one line
(b) A statement that contains other statements
(c) A statement that contains at least one arithmetic expression

3. Which of the following statements are true regarding headers in Python?
(a) Headers begin with a keyword and end with a colon.
(b) Headers always occur in pairs.
(c) All headers of the same compound statement must be indented the same amount.

4. Which of the following statements is true?
(a) Statements within a suite can be indented a different amount.
(b) Statements within a suite can be indented a different amount as long as all headers in the

statement that it occurs in are indented the same amount.
(c) All headers must be indented the same amount as all other headers in the same statement,
and all statements in a given suite must be indented the same amount.

5 . The elif header allows for,
(a) Multi-way selection that cannot be accomplished otherwise
(b) Multi-way selection as a single if statement
(c) The use of a “catch-all” case in multi-way selection

ANSWERS: 1. False, 2. (b), 3. (a) (c), 4. (c), 5. (b)

3.4 Iterative Control

An **iterative control statement** is a control statement providing the repeated execution of a set of instructions. An *iterative control structure* is a set of

instructions and the iterative control statement(s) controlling their execution. Because of their repeated execution, iterative control structures are commonly referred to as “loops.” We look at one specific iterative control statement next, the while statement.

An **iterative control statement** is a control statement that allows for the repeated execution of a set of statements.

3.4.1 While Statement

A **while statement** is an iterative control statement that repeatedly executes a set of statements based on a provided Boolean expression (condition). All iterative control needed in a program can be achieved by use of the while statement. Figure 3-17 contains an example of a while loop in Python that sums the first n integers, for a given (positive) value n entered by the user.

while statement	Example use

`while condition:`
`  suite`
	`sum = 0`
	`current = 1`
	
	`n = int(input('Enter value: '))`
	
	`while current <= n:`
	`  sum = sum + current`
	`  current = current + 1`

FIGURE 3-17 The while Statement in Python

As long as the condition of a while statement is true, the statements within the loop are (re)executed. Once the condition becomes false, the iteration terminates and control continues with the first statement after the while loop. Note that it is possible that the first time a loop is reached, the condition may be false, and therefore the loop would never be executed.

Suppose, for the example in the figure, that the user enters the value 3. Since variable current is initialized to 1 (referred to as a *counter variable*), the first time the while statement is reached, current ,53 is true. Thus, the statements within the loop are executed and sum is u pdated to sum 1current. Since sum is initialized to 0, sum becomes 1. Similarly, current is updated and assigned to 2. After the first time through the loop, control returns to the “top” of the loop. The condition is again found to be true and thus the loop is executed a second time.

In this iteration, both sum and current become 3. In the next iteration, the condition is still true, and therefore, the loop is executed a third time. This time, sum becomes 6 and current becomes 4. Thus, when control returns to the top of the loop, the condition is False and the loop terminates. The final value of sum therefore is 6(1 1 2 1 3). This process is summarized in Figure 3-18.

Iteration	sum	current	current <= 3	sum = sum + current	current = current + 1
1	0	1	True	sum = 0 + 1 (1)	current = 1 + 1 (2)
2	1	2	True	sum = 1 + 2 (3)	current = 2 + 1 (3)
3	3	3	True	sum = 3 + 3 (6)	current = 3 + 1 (4)
4	6	4	False	loop termination	

FIGURE 3-18 Iterative Steps for Adding First Three Integers

A **while statement** is an iterative control statement that repeatedly executes a set of statements based on a provided Boolean expression.

3.4.2 Input Error Checking

The while statement is well suited for input error checking in a program. This is demonstrated in the revised version of the temperature conversion program from Figure 3-14, reproduced in Figure 3-19.

```
# Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)

# Display program welcome
print('This program will convert temperatures (Fahrenheit/Celsius)')
print('Enter (F) to convert Fahrenheit to Celsius')
print('Enter (C) to convert Celsius to Fahrenheit')

# Get temperature to convert
which = input('Enter selection: ')

while which != 'F' and which != 'C':
   which = input("Please enter 'F' or 'C': ")

temp = int(input('Enter temperature to convert: '))

# Determine temperature conversion needed and display results
if which == 'F':
    converted_temp = format((temp - 32) * 5/9, '.1f')
    print(temp, 'degrees Fahrenheit equals', converted_temp, 'degrees Celsius')
else:
    converted_temp = format((9/5 * temp) + 32, '.1f')
    print(temp, 'degrees Celsius equals', converted_temp, 'degrees Fahrenheit')
```

FIGURE 3-19 Temperature Conversion Program (Invalid Input Checking)

The difference in this program from the previous version is that rather than terminating on invalid input, the program continues to prompt the user until a valid temperature conversion, 'F' or 'C', is entered. Thus, the associated input

statement is contained within a while loop that keeps iterating as long as variable which contains an invalid value. Once the user enters a proper value, the loop terminates allowing the program to continue.

LET’S TRY IT
In IDLE, create and run a simple program containing the code below and observe the results. Make sure to indent the code exactly as shown.

```
n 5 10
sum 5 0
current 5 1

while current < 5 n:
sum sum 1 current current current 1 1

print(sum)
???
n 5 10
sum 5 0
current 5 1
while current <5 n:

sum sum 1 current current current 1 1 print(sum)
???
```

The while statement is well suited for input error checking.

3.4.3 Infinite loops

An **infinite loop** is an iterative control structure that never terminates (or eventually terminates with a system error). Infinite loops are generally the result of programming errors. For example, if the condition of a while loop can never be false, an infinite loop will result when executed. Consider if the program segment in Figure 3-17, reproduced in Figure 3-20, omitted the statement incrementing variable current. Since current is initialized to 1, it would remain 1 in all iterations, causing the expression current ,5n to be always be true. Thus, the loop would never terminate.

```
# add up first n integers
sum = 0
current = 1

n = int(input('Enter value: '))

while current <= n:
    sum = sum + current
```

FIGURE 3-20 Infinite Loop

Such infinite loops can cause a program to “hang,” that is, to be unresponsive to the user. In such cases, the program must be terminated by use of some special keyboard input (such as ctrl-C) to interrupt the execution.

LET’S TRY IT From IDLE, create and run a simple program containing the code below and observe the results. Make sure to indent the code exactly as shown.
To terminate an executing loop, hit ctrl-C.
while True:
print ('Looping') ???

n 5 10
sum 5 0
current 5 1

while current ,5 n: sum sum 1 current print(sum)

???
n 5 10
sum 5 0
current 5 1

while current ,5 n: sum sum 1 current n n 2 1

print(sum)
???
An **infinite loop** is an iterative control structure that never terminates (or eventually terminates with a system error).
3.4.4 Definite vs. Indefinite Loops
A **definite loop** is a program loop in which the number of times the loop will iterate can be determined before the loop is executed. For example, the while

loop introduced in Figure 3-17 is a definite loop,

```
sum 5 0
current 5 1
n 5 input('Enter value: ')
while current ,5 n:

sum 5 sum 1 current
current 5 current 1 1
```

Although it is not known what the value of n will be until the input statement is executed, its value *is* known by the time the while loop is reached. Thus, it will execute “n times.”

An **indefinite loop** is a program loop in which the number of times that the loop will iterate cannot be determined before the loop is executed. Consider the while loop in the temperature conversion program of Figure 3-19.

```
which 5 input("Enter selection: ")
while which ! 5 'F' and which ! 5 'C':
which 5 input("Please enter 'F' or 'C': ")
```

In this case, the number of times that the loop will be executed depends on how many times the user mistypes the input. Thus, a while statement can be used to construct both definite and indefinite loops. In the next chapter we look at the for statement, specifically suited for the construction of definite loops.

A **definite loop** is a program loop in which the number of times the loop will iterate can be determined before the loop is executed. A **indefinite loop** is a program loop in which the number of times the loop will iterate is not known before the loop is executed.

3.4.5 Boolean Flags and Indefinite Loops

Often the condition of a given while loop is denoted by a single Boolean variable, called a **Boolean flag**. This is shown in Figure 3-21.

Boolean variable valid_entries is a Boolean flag, controlling the while loop at **line 12**. If the mileage of the last oil change is greater than the current mileage, an error message is displayed (**lines 17–18**), and the while loop is re-executed. If the current mileage is greater than (or equal to) the mileage of the last oil change, miles_traveled is set to this difference and valid_ entries is set to True,

causing the loop to terminate. Thus, **lines 23–28** display either that they are due for an oil change, an oil change will soon be needed, or there is no immediate need for an oil change.

A single Boolean variable used as the condition of a given control statement is called a **Boolean flag**.

```
1  # Oil Change Notification Program
2
3  # display program welcome
4  print('This program will determine if your car is in need of an oil change')
5
6  # init
7  miles_between_oil_change = 7500  # num miles between oil changes
8  miles_warning = 500      # how soon to warn of needed oil change
9  valid_entries = False
10
11 # get mileage of last oil change and current mileage and display
12 while not valid_entries:
13     mileage_last_oilchange = int(input('Enter mileage of last oil change: '))
14     current_mileage = int(input('Enter current mileage: '))
15
16     if current_mileage < mileage_last_oilchange:
17         print('Invalid entry - current mileage entered is less than')
18         print('mileage entered of last oil change')
19     else:
20         miles_traveled = current_mileage - mileage_last_oilchange
21         valid_entries = True
22
23 if miles_traveled >= miles_oil_change:
24     print('You are due for an oil change')
25 elif miles_traveled >= miles_oil_change - miles_warning:
26     print('You will soon be due for an oil change')
27 else:
28     print('You are not in immediate need of an oil change')
```

FIGURE 3-21 Indefinite Loop Using a Boolean Flag

3.4.6 Let’s Apply It—Coin Change Exercise Program

The Python program in Figure 3-22 implements an exercise for children learning to count change. It displays a random value between 1 and 99 cents, and asks the user to enter a set of coins that sums exactly to the amount shown. The program utilizes the following programming features:

➤ while loop ➤ if statement ➤ Boolean flag ➤ random number generator

On **line 3**, the random module is imported for use of function randint. This function is called (on **line 18**) to randomly generate a coin value for the user to match, stored in variable amount. **Lines 6–10** provide the program greeting. On **line 13** variable terminate is initialized to False, used to control when the main loop (and thus the program) terminates. On **line 14**, empty_str is initialized to

the empty string literal '', used to determine when the user has entered an empty line to end the coin entries. These two variables need only be initialized once, and therefore are assigned before the main while loop.

The game begins on **line 17**. Since Boolean flag terminate is initialized to False, the while loop is executed. Besides variable amount, game_over is initialized to False, and total is initialized to 0. Variable game_over serves as another Boolean flag to determine if the current game is to continue or not. The coin entry ends if either the user enters a blank line (indicating that they are done entering coins) in which case the result is displayed and game_over is set to True (**line 37–43**), or if the total amount accumulated exceeds the total amount to be matched (on **line 45–48**).

At the top of the while loop, a third Boolean flag is used, valid_entry. The value of this flag determines whether the user should be prompted again because of an invalid input—a value

Program Execution ...

```
The purpose of this exercise is to enter a number of coin values
that add up to a displayed target value.

Enter coins values as 1-penny, 5-nickel, 10-dime and 25-quarter.
Hit return after the last entered coin value.
----------------
Enter coins that add up to 63 cents, one per line.

Enter first coin: 25
Enter next coin: 25
Enter next coin: 10
Enter next coin:
Sorry - you only entered 60 cents.

Try again (y/n)?: y
Enter coins that add up to 21 cents, one per line.

Enter first coin: 11
Invalid entry
Enter next coin: 10
Enter next coin: 10
Enter next coin: 5
Sorry - total amount exceeds 21 cents.

Try again (y/n)?: y
Enter coins that add up to 83 cents, one per line.

Enter first coin: 25
Enter next coin: 25
Enter next coin: 25
Enter next coin: 5
Enter next coin: 1
Enter next coin: 1
Enter next coin: 1
Enter next coin:
Correct!

Try again (y/n)?: n
Thanks for playing ... goodbye
```

FIGURE 3-22 Coin Change Exercise Program (*Continued*)

other than '1', '5', '10', or '25', or the empty string. Note the use of the membership operator in (**line 32**). Thus, once the user inputs an appropriate value, valid_entry is set to True (**line 33**)—otherwise, the message 'Invalid entry' is displayed and valid_entry remains False, causing the loop to execute again. The list of valid entered values on line 32 includes variable empty_str

since this is the value input when the user hits return to terminate their entry of coin values. When the empty string is found (**line 37**), the total coin value entered in variable total is compared with variable amount (the amount to be matched). If equal, the message 'Correct!' is displayed (**line 39**)—otherwise, a message is displayed indicating how much they entered. This amount is always less than the required amount, since whenever variable total exceeds amount, the current game ends (**lines 46–48**).

```python
# Coin Change Exercise Program

import random

# program greeting
print('The purpose of this exercise is to enter a number of coin values')
print('that add up to to a displayed target value.\n')
print('Enter coins values as 1-penny, 5-nickel, 10-dime and 25-quarter')
print("Hit return after the last entered coin value.")
print('----------------')

# init
terminate = False
empty_str = ''

# start game
while not terminate:
    amount = random.randint(1,99)
    print('Enter coins that add up to', amount, 'cents, one per line.\n')
    game_over = False
    total = 0

    while not game_over:
        valid_entry = False

        while not valid_entry:
            if total == 0:
                entry = input('Enter first coin: ')
            else:
                entry = input('Enter next coin: ')

            if entry in (empty_str,'1','5','10','25'):
                valid_entry = True
            else:
                print('Invalid entry')

        if entry == empty_str:
            if total == amount:
                print('Correct!')
            else:
                print('Sorry - you only entered', total, 'cents.')

            game_over = True
        else:
            total = total + int(entry)
            if total > amount:
                print('Sorry - total amount exceeds', amount, 'cents.')
                game_over = True

        if game_over:
            entry = input('\nTry again (y/n)?: ')

            if entry == 'n':
                terminate = True

print('Thanks for playing ... goodbye')
```

FIGURE 3-22 Coin Change Exercise Program

When **line 50** is reached, Boolean flag game_over may be either True or False. It is True when the user has indicated that they have entered all their coin values (by hitting return), or if the total of the coin values entered exceeds the value in variable amount—it is False otherwise. Therefore, flag variable game_over is used to determine whether the user should be prompted to play another game (**line 51**). If they choose to quit the program when prompted, then Boolean variable terminate is set to True. This causes the encompassing while loop at **line 17** to terminate, leaving only the final "goodbye" message on **line 56** to be executed before the program terminates.

Self-T est Questions
1. A while loop continues to iterate until its condition becomes false. TRUE/FALSE
2. A while loop executes zero or more times. TRUE/FALSE
3. All iteration can be achieved by a while loop. TRUE/FALSE

4. An infinite loop is an iterative control structures that,
(a) Loops forever and must be forced to terminate
(b) Loops until the program terminates with a system error
(c) Both of the above

5. The terms *definite loop* and *indefinite loop* are used to indicate whether,
(a) A given loop executes at least once
(b) The number of times that a loop is executed can be determined before the loop is executed. **(c)** Both of the above

6. A Boolean flag is,
(a) A variable
(b) Has the value True or False
(c) Is used as a condition for control statements
(d) All of the above

ANSWERS: 1. True, 2. True, 3. True, 4. (c), 5. (b), 6. (d)

COMPUTATIONAL PROBLEM SOLVING
3.5 Calendar Month Program

3.5.1 The Problem

The problem is to display a calendar month for any given month between

January 1800 and December 2099. The format of the month should be as shown in Figure 3-23.

3.5.2 Problem Analysis

T wo specific algorithms are needed for this problem. First, we need an algorithm for computing the first day of a given month for years 1800 through 2099. This algorithm is given in Chapter 1. The second needed algorithm is for appropriately displaying the calendar month, given the day of the week that

MAY 2012

Sun	Mon	Tues	Wed	Thur	Fri	Sat
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

FIGURE 3-23 Calendar Month Display

the first day falls on, and the number of days in the month. We shall develop this algorithm. The data representation issues for this problem are straight forward.

3.5.3 Program Design
Meeting the Program Requirements

We will develop and implement an algorithm that displays the month as given. There is no requirement of how the month and year are to be entered. We shall therefore request the user to enter the month and year as integer values, with appropriate input error checking.

Data Description

What needs to be represented in the program is the month and year entered, whether the year is a leap year or not, the number of days in the month, and which day the first of the month falls on. Given that information, the calendar month can be displayed. The year and month will be entered and stored as integer values, represented by variables year and month,

year 52012 month 5 5

The remaining values will be computed by the program based on the given year and month, as given below,
leap_year num_days_in_month day_of_week
Variable leap_year holds a Boolean (True/False) value. Variables num_days_in_month and day_of_week each hold integer values.
Algorithmic Approach
First, we need an algorithm for determining the day of the week that a given date falls on. The algorithm for this from Chapter 1 is reproduced in Figure 3-24.

We also need to determine how many days are in a given month, which relies on an algorithm for determining leap years for the month of February. The code for this has already been developed in the “Number of Days in Month” program in section 3.3.4. We shall also reuse the portion of code from that program for determining leap years, reproduced below.

if (year % 4 55 0) and (not (year % 100 55 0) or year % 400): leap_year 5 True
else:
leap_Year 5 False

Let’s review how this algorithm works, and try to determine the day of the week on which May 24, 2025 falls. First, variable century_digits (holding the first two digits of the year) is set to 20 and year_digits (holding the last two digits of the year) is set to 25 (**steps 1 and 2**). Variable value, in **step 3**, is then set to

value 5 year_digits 1 floor(year_digits / 4)
5 25 1 floor(25/4) → 25 1 floor(6.25) → 25 1 6 → 31

To determine the day of the week for a given **month**, **day**, and **year**:

1. Let **century_digits** be equal to the first two digits of the year.
2. Let **year_digits** be equal to the last two digits of the year.
3. Let **value** be equal to **year_digits** + floor(**year_digits** / 4)
4. If **century_digits** equals 18, then add 2 to **value**, else
   if **century_digits** equals 20, then add 6 to **value**.
5. If the **month** is equal to January and the **year** is not a leap year,
   then add 1 to **value**, else,

   if the **month** is equal to February and the **year** is a leap year, then
   add 3 to **value**; if not a leap year, then add 4 to **value,** else,

   if the **month** is equal to March or November, then add 4 to **value**, else,

   if the **month** is equal to April or July, then add 0 to **value**, else,

   if the **month** is equal to May, then add 2 to **value**, else,

   if the **month** is equal to June, then add 5 to **value**, else,

   if the **month** is equal to August, then add 3 to **value**, else,

   if the **month** is equal to October, then add 1 to **value**, else,

   if the **month** is equal to September or December, then add 6 to **value**,
6. Set **value** equal to (**value** + **day**) mod 7.
7. If **value** is equal to 1, then the day of the week is Sunday; else

   if **value** is equal to 2, day of the week is Monday; else

   if **value** is equal to 3, day of the week is Tuesday; else

   if **value** is equal to 4, day of the week is Wednesday; else

   if **value** is equal to 5, day of the week is Thursday; else

   if **value** is equal to 6, day of the week is Friday; else

   if **value** is equal to 0, day of the week is Saturday

FIGURE 3-24 Day of the Week Algorithm (from Chapter 1)

In **step 4**, since century_digits is equal to 20, value is incremented by 6,
value → 6 → 37

In **step 5**, since the month is equal to May, value is incremented by 2,
value → 2 → 39

In **step 6**, value is updated based on the day of the month. Since we want to determine the day of the week for the 24th (of May), value is updated as follows,

value 5 (value 1 *day of the month*) mod 7
5 (39 1 24) mod 7
5 63 mod 7

Therefore, by **step 7** of the algorithm, the day of the week for May 24, 2025 is a Saturday. A table for the interpretation of the day of the week for the final computed value is given in Figure 3-25.

1	2	3	4	5	6	0
Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday

FIGURE 3-25 Interpretation of the Day of the Week Algorithm Results

Overall Program Steps

The overall steps in this program design are given in Figure 3-26.

3.5.4 Program Implementation and Testing

Stage 1—Determining the Number of Days in the Month/Leap Years

We develop and test the program in three stages. First, we implement and test the code that determines, for a given month and year, the number of days in the month and whether the year is a leap year or not, given in Figure 3-27.

FIGURE 3-26 Overall Steps of Calendar
Month Program

```
1   # Calendar Month Program (stage 1)
2
3   # init
4   terminate = False
5
6   # program greeting
7   print('This program will display a calendar month between 1800 and 2099')
8
9   while not terminate:
10      # get month and year
11      month = int(input('Enter month 1-12 (-1 to quit): '))
12
13      if month == -1:
14          terminate = True
15      else:
16          while month < 1 or month > 12:
17              month = int(input('INVALID INPUT - Enter month 1-12: '))
18
19          year = int(input('Enter year (yyyy): '))
20
21          while year < 1800 or year > 2099:
22              year = int(input('INVALID - Enter year (1800-2099): '))
23
24          # determine if leap year
25          if (year % 4 == 0) and (not (year % 100 == 0) or (year % 400 == 0)):
26              leap_year = True
27          else:
28              leap_year = False
29
30          # determine num of days in month
31          if month in (1, 3, 5, 7, 8, 10, 12):
32              num_days_in_month = 31
33          elif month in (4, 6, 9, 11):
34              num_days_in_month = 30
35          elif leap_year:  # February
36              num_days_in_month = 29
37          else:
38              num_days_in_month = 28
39
40          print ('\n', month, ',', year, 'has', num_days_in_month, 'days')
41
42          if leap_year:
43              print (year, 'is a leap year\n')
44          else:
45              print (year, 'is NOT a leap year\n')
```

FIGURE 3-27 First Stage of Calendar Month Program

The month and year entered by the user are stored in variables month and year. While loops are used at **lines 16** and **21** to perform input error checking. **Lines 25–28** are adapted from the previous Number of Days in Month program for determining leap years. **Lines 31–38** are similar to the previous program for determining the number of days in a month, stored in variable num_days_in_month. **Lines 42– 45** contain added code for the purpose of testing. These instructions will not be part of the final program. The program

continues to prompt for another month until 21 is entered. Thus, Boolean flag terminate is initialized to False (**line 4**) and set to True (**line 14**) when the program is to terminate.

Stage 1 Testing

We give output from the testing of this version of the program in Figure 3-28.

```
Enter month (1-12): 14
INVALID INPUT
Enter month (1-12): 1
Enter year (yyyy): 1800

1 , 1800 has 31 days
1800 is NOT a leap year
```

FIGURE 3-28 Example Output of First Stage Testing

The set of test cases for this stage of the program is given in Figure 3-29. The test cases are selected such that each month is tested within the 1800s, 1900s, and 2000s. The month of February has a number of test cases to ensure that the program is working for non-leap years (1985), “typical” leap years (1984), and exception years (1900 and 2000). The test plan also includes the “extreme” cases of January 1800 and December 2099 (the beginning and end of the range of valid months). All test cases are shown to have passed, and thus we can move on to stage 2 of the program development.

Calendar Month	Expected Results num days	Expected Results leap year	Actual Results num days	Actual Results leap year	Evaluation
January 1800	31	no	31	no	Passed
February 1900	28	no	28	no	Passed
February 1984	29	yes	29	yes	Passed
February 1985	28	no	28	no	Passed
February 2000	29	yes	29	yes	Passed
March 1810	31	no	31	no	Passed
April 1912	30	yes	30	yes	Passed
May 2015	31	no	31	no	Passed
June 1825	30	no	30	no	Passed
July 1928	31	yes	31	yes	Passed
August 2031	31	no	31	no	Passed
September 1845	30	no	30	no	Passed
October 1947	31	no	31	no	Passed
November 2053	30	no	30	no	Passed
December 2099	31	no	31	no	Passed

FIGURE 3-29 Results of Execution of Test Plan for Stage 1

Stage 2—Determining the Day of the Week

We give the next stage of the program in Figure 3-30. This version includes the code for determining the day of the week for the first day of a given month and year (**lines 40–71**), with the final print statement (**line 74**) displaying the test results. Note that for testing purposes, there is no need to convert the day number into the actual name (e.g., “Monday”)—this “raw output” is good enough. Also, for this program, we will need to determine only the day of the week for the first day of any

```
# Calendar Month Program (stage 2)

# init
terminate = False

# program greeting
print('This program will display a calendar month between 1800 and 2099'

while not terminate:
    # get month and year
    month = int(input('Enter month (1-12): '))

    if month == -1:
        terminate = True
    else:
        while month < 1 or month > 12:
            month = int(input('INVALID INPUT - Enter month 1-12: '))

        year = int(input('Enter year (yyyy): '))

        while year < 1800 or year > 2099:
            year = int(input('INVALID - Enter year (1800-2099): '))

        # determine if leap year
        if (year % 4 == 0) and (not(year % 100 == 0) or (year % 400 == 0)):
            leap_year = True
        else:
            leap_year = False

        # determine num of days in month
        if month in (1,3,5,7,8,10,12):
            num_days_in_month = 31
        elif month in (4,6,9,11):
            num_days_in_month = 30
        elif leap_year:  # February
            num_days_in_month = 29
        else:
            num_days_in_month = 28

        # determine day of the week
        century_digits = year // 100
        year_digits = year % 100

        value = year_digits + (year_digits // 4)

        if century_digits == 18:
            value = value + 2
        elif century_digits == 20:
            value = value + 6

```

FIGURE 3-30 Second Stage of Calendar Month Program (*Continued*)

given month, since all remaining days follow sequentially. Therefore, the day value in the day of the week algorithm part of the code is hard-coded to 1 (on **line 71**). Let's look at the code that implements the day of the week algorithm.

The algorithm operates separately on the first two digits and last two digits of the year. On **line 41**, integer division is used to extract the first two digits of the year (for example, 1860 // 100

```
51          if month == 1 and not leap_year:
52              value = value + 1
53          elif month == 2:
54              if leap_year:
55                  value = value + 3
56              else:
57                  value = value + 4
58          elif month == 3 or month == 11:
59              value = value + 4
60          elif month == 5:
61              value = value + 2
62          elif month == 6:
63              value = value + 5
64          elif month == 8:
65              value = value + 3
66          elif month == 9 or month == 12:
67              value = value + 6
68          elif month == 10:
69              value = value + 1
70
71          day_of_week = (value + 1) % 7  # 1-Sunday, 2-Monday, ...,
72
73          # display results
74          print('Day of the week is', day_of_week)
```

FIGURE 3-30 Second Stage of Calendar Month Program

equals 18). On **line 42**, the modulus operator, %, is used to extract the last two digits (for example, 1860 % 100 equals 60). The rest of the program (through **line 71**) follows the day of the week algorithm given above.

Stage 2 Testing

We give a sample test run of this version of the program in Figure 3-31.

```
1
Enter month (1-12): 4
Enter year (yyyy): 1860
Day of the week is 1
Enter month (1-12): -1
>>>
```

FIGURE 3-31 Example Output of Second Stage Testing

Figure 3-32 shows the results of the execution of the test plan for this version of the program. It includes the same months as in the test plan for the first stage. Since all test cases passed, we can move on to the final stage of program development.

Final Stage—Displaying the Calendar Month

In the final stage of the program (Figure 3-33), we add the code for displaying the calendar month. The corresponding name for the month number is determined on **lines 74–97** and displayed
(**line 100**). The while loop at **line 113** moves the cursor to the proper starting column by “printing”

Calendar Month	Expected Results first day of month	Actual Results first day of month	Evaluation
January 1800	4 (Wednesday)	4	Passed
February 1900	5 (Thursday)	5	Passed
February 1984	4 (Wednesday)	4	Passed
February 1985	6 (Friday)	6	Passed
February 2000	3 (Tuesday)	3	Passed
March 1810	5 (Thursday)	5	Passed
April 1912	2 (Monday)	2	Passed
May 2015	6 (Friday)	6	Passed
June 1825	4 (Wednesday)	4	Passed
July 1928	1 (Sunday)	1	Passed
August 2031	6 (Friday)	6	Passed
September 1845	2 (Monday)	2	Passed
October 1947	4 (Wednesday)	4	Passed
November 2053	0 (Saturday)	0	Passed
December 2099	3 (Tuesday)	3	Passed

FIGURE 3-32 Results of Execution of Test Plan for Stage 2

```
# Calendar Month Program

# init
terminate = False

# program greeting
print('This program will display a calendar month between 1800 and 2099')

while not terminate:
    # get month and year
    month = int(input('Enter month 1-12 (-1 to quit): '))

    if month == -1:
        terminate = True
    else:
        while month < 1 or month > 12:
            month = int(input('INVALID - Enter month (1-12): '))

        year = int(input('Enter year (yyyy): '))

        while year < 1800 or year > 2099:
            year = int(input('INVALID - Enter year (1800-2099): '))

```

FIGURE 3-33 Final Stage of Calendar Month Program (*Continued*)

```
24          # determine if leap year
25          if (year % 4 == 0) and (not(year % 100 == 0) or (year % 400 == 0)):
26              leap_year = True
27          else:
28              leap_year = False
29
30          # determine num of days in month
31          if month in (1,3,5,7,8,10,12):
32              num_days_in_month = 31
33          elif month in (4,6,9,11):
34              num_days_in_month = 30
35          elif leap_year:  # February
36              num_days_in_month = 29
37          else:
38              num_days_in_month = 28
39
40          # determine day of the week
41          century_digits = year // 100
42          year_digits = year % 100
43
44          value = year_digits + (year_digits // 4)
45
46          if century_digits == 18:
47              value = value + 2
48          elif century_digits == 20:
49              value = value + 6
50
51          if month == 1 and not leap_year:
52              value = value + 1
53          elif month == 2:
54              if leap_year:
55                  value = value + 3
56              else:
57                  value = value + 4
58          elif month == 3 or month == 11:
59              value = value + 4
60          elif month == 5:
61              value = value + 2
62          elif month == 6:
63              value = value + 5
64          elif month == 8:
65              value = value + 3
66          elif month == 9 or month == 12:
67              value = value + 6
68          elif month == 10:
69              value = value + 1
70
71          day_of_week = (value + 1) % 7  # 1-Sun, 2-Mon, ..., 0-Sat
72
```

FIGURE 3-33 Final Stage of Calendar Month Program (*Continued*)

the column_width number of blank characters (4) for each column to be skipped. The while loop at **line 119** displays the dates. Single-digit dates are output (**line 121**) with three leading spaces, and two-digit dates with two (**line 123**) so that the columns line up. Each uses the **newline suppression form of print**, print(..., end 5") to prevent the cursor from moving to the next screen line until it is time to do so.

Variable current_day is incremented from 1 to the number of days in the month. Variable current_col is also incremented by 1 to keep track of what column the current date is being

```
        # determine month name
        if month == 1:
            month_name = 'January'
        elif month == 2:
            month_name = 'February'
        elif month == 3:
            month_name = 'March'
        elif month == 4:
            month_name = 'April’
        elif month == 5:
            month_name = 'May'
        elif month == 6:
            month_name = 'June'
        elif month == 7:
            month_name = 'July'
        elif month == 8:
            month_name = 'August'
        elif month == 9:
            month_name = 'September'
        elif month == 10:
            month_name = 'October'
        elif month == 11:
            month_name = 'November'
        else:
            month_name = 'December'

        # display month and year heading
        print('\n', ' ' + month_name, year)

        # display rows of dates
        if day_of_week == 0:
            starting_col = 7
        else:
            starting_col = day_of_week

        current_col = 1
        column_width = 4
        blank_char = ' '
        blank_column = format(blank_char, str(column_width))

        while current_col <= starting_col:
            print(blank_column, end='')
            current_col = current_col + 1

        current_day = 1

        while current_day <= num_days_in_month:
            if current_day < 10:
                print (format(blank_char, '3') + str(current_day), end='')
            else:
                print (format(blank_char, '2') + str(current_day), end='')

            if current_col <= 7:
                current_col = current_col + 1
            else:
                current_col = 1
                print()

            current_day = current_day + 1

        print('\n')
```

FIGURE 3-33 Final Stage of Calendar Month Program

displayed in. When current_col equals 7, it is reset to 1 (**line 128**) and print() moves the cursor to the start of the next line (**line 129**). Otherwise, current_col is simply incremented by 1 (**line 126**).

An example test run of this final version of the program is given in Figure 3-34.

```
This program will display a calendar month between 1800 and 2099

Enter month 1-12 (-1 to quit): 1
Enter year (yyyy): 1800

  January   1800
                    1   2   3   4
   5   6   7   8   9  10  11  12
  13  14  15  16  17  18  19  20
  21  22  23  24  25  26  27  28
  29  30  31

Enter month (1-12): -1
>>>
```

FIGURE 3-34 Example Output of Final Stage Testing

Something is obviously wrong . The calendar month is displayed with eight columns instead of seven. The testing of all other months produces the same results. Since the first two stages of the program were successfully tested, the problem must be in the code added in the final stage. The code at **line 74** simply assigns the month name. Therefore, we reflect on the logic of the code starting on **line 103**.

Lines 128–129 is where the column is reset back to column 1 and a new screen line is started, based on the current value of variable current_col,

if current_col ,5 7:
current_col 5 current_col 1 1
else:
current_col 5 **1**
print()

V ariable current_col is initialized to 1 at **line 108**, and is advanced to the proper starting column on **lines 113–115**. Variable starting_col is set to the value (0-6) for the day of the week for the particular month being displayed. Since the day

of the week results have been successfully tested, we can assume that current_col will have a value between 0 and 6. With that assumption, we can step though **lines 125–129** and see if this is where the problem is. Stepping through a program on paper by tracking the values of variables is referred to as **deskchecking**. We check what happens as the value of current_col approaches 7, as shown in Figure 3-35.

Now it is clear what the problem is—the classic *"off by one" error*! The condition of the while loop should be current_col ,7, *not*current_col ,57. Current_col should be reset to 1 once the seventh column has been displayed (when current_col is 7). Using the

Current value of `current_col`	Value of condition `current_col <= 7`	Updated value of `current_col`
5	True	6
6	True	7
7	True	8
8	False	1
1	True	2
etc.		

FIGURE 3-35 Deskchecking the Value of Variable current_col

,5 operator causes current_col to be reset to 1 only after an *eighth* column is displayed. Thus, we make this correction in the program,

if current_col , 7:
current_col 5 current_col 1 1
else:
current_col 5 1
print()

After re-executing the program with this correction we get the current output, depicted in Figure 3-36.

```
This program will display a calendar month between 1800 and 2099

Enter month 1-12 (-1 to quit): 1
Enter year (yyyy): 1800

  January   1800
                    1   2   3
   4   5   6   7   8   9  10
  11  12  13  14  15  16  17
  18  19  20  21  22  23  24
  25  26  27  28  29  30  31

Enter month (1-12): -1
>>>
```

FIGURE 3-36 Display Output of Final Stage of Calendar Month Program

Although the column error has been corrected, we find that the first of the month appears under the wrong column—the month should start on a Wednesday (fourth column), not a Thursday column (fifth column). The problem must be in how the first row of the month is displayed. Other months are tested, each found to be off by one day. We therefore look at **lines 113–115** that are responsible for moving over the cursor to the correct starting column,

while current_col ,5 starting_col:
print(blank_column, end 5")
current_col 5 current_col 1 1

We consider whether there is another “off by one” error. Reconsidering the condition of the while loop, we realize that, in fact, this is the error. If the correct starting column is 4 (Wednesday), then the cursor should move past three columns and place a 1 in the fourth column. The current condition, however, would move the cursor past *four* columns, thus placing a 1 in the fifth column (Thursday). The corrected code is given below.

while current_col , starting_col: print(' ', end ")
current_col 5 current_col 1 1

The month is now correctly displayed. We complete the testing by executing the program on a set of test cases (Figure 3-37). Although the test plan is not as complete as it could be, it includes test cases for months from each century,

including both leap years and non-leap years.

Calendar Month	Expected Results first day of month	num days	Evaluation
April 1912	Sunday	30	Passed
February 1985	Monday	28	Passed
May 2015	Tuesday	31	Passed
January 1800	Wednesday	31	Passed
February 1900	Thursday	28	Passed
February 1984	Friday	29	Passed
January 2011	Saturday	31	Passed

FIGURE 3-37 Results of Execution of Test Plan for Final Stage

CHAPTER SUMMARY General Topics

Control Statement/Control Structure
Sequential, Selection, and Iterative Control Relational Operators/Boolean Operators/

Boolean Expressions
Operator Precedence and Boolean Expressions Logically Equivalent Boolean Expressions Short-Circuit (Lazy) Evaluation
Selection Control Statements/if Statement Compound Statement
Multi-way Selection
While Statements
Input Error Checking
Infinite Loops
Definite vs. Indefinite Loops
Boolean Flags and Indefinite Loops
Deskchecking

Python-Specific Programming Topics

Membership Operators in, not in
if Statement in Python/else and elif headers Indentation in Python

Multi-way Selection in Python
while Statement in Python

CHAPTER EXERCISES

Section 3.1
1. Which of the three forms of control is an implicit form of control?
2. What is meant by a “straight-line” program?
3. What is the difference between a control statement and a control structure?

Section 3.2

4. The Boolean data type contains two literal values, denoted as
_____ and _____ in Python.
5. Which of the following relational expressions evaluate to True?
(a) 5 , 8 **(c)** '10' , '8'
(b)'5' , '8' **(d)** 'Jake' , 'Brian'
6. Which of the following relational expressions evaluate to False?
(a) 5 ,5 5 **(c)** 5 55 5 **(e)** 5 !5 10
(b) .5 5 **(d)** 5 !5 5
7. Give an appropriate expression for each of the following.
(a) To determine if the number 24 does *not* appear in a given list of numbers assigned to variable nums. **(b)** To determine if the name 'Ellen' appears in a list of names assigned to variable names. **(c)** To determine if a single last name stored in variable last_name is either 'Morris' or 'Morrison'.
8. Evaluate the following Python expressions.
(a) (12 * 2) 55 (3 * 8)
(b)(14 * 2) !5 (3 * 8)
9. What value for x makes each of the following Boolean expressions true?
(a) x or False
(b)x and True
(c) not (x or False)
(d)not (x and True)
10. Evaluate the Boolean expressions below for n 5 10 and k 5 20.
(a) (n . 10) and (k 55 20)
(b)(n . 10) or (k 55 20)
(c) not((n . 10) and (k 55 20))
(d)not(n . 10) and not(k 55 20)
(e) (n .10) or (k 5510 or k !5 5)

11. Give an appropriate Boolean expression for each of the following.
(a) Determine if variable num is greater than or equal to 0, and less than 100.
(b) Determine if variable num is less than 100 and greater than or equal to 0, or it is equal to 200. **(c)** Determine if either the name 'Thompson' or 'Wu' appears in a list of names assigned to variable last_names.
(d) Determine if the name 'Thomson' appears and the name 'Wu' does not appear in a list of last names assigned to variable last_names.
12. Evaluate the following Boolean expressions for num1 5 10 and num2 5 20.
(a) not (num1 ,1) and num2 , 10
(b)not (num1 , 1) and num2 , 10 or num1 1 num3 , 100

13. Give a logically equivalent expression for each of the following. **(a)** num !5 25 or num 55 0
(b)1 ,5 num and num ,5 50
(c) not num . 100 and not num , 0
(d)(num , 0 or num . 100)

Section 3.3
14. Give an appropriate if statement for each of the following.
(a) An if statement that displays 'within range' if num is between 0 and 100, inclusive. **(b)** An if statement that displays 'within range' if num is between 0 and 100, inclusive, and displays 'out of range' otherwise.

15. Rewrite the following if-else statements using a single if statement and elif headers.

```
if temperature . 5 85 and humidity . 60: print('muggy day today')
else:
if temperature . 5 85:
print('warm, but not muggy today') else:
if temperature . 5 65:
print('pleasant today')
else:
if temperature , 5 45:
print('cold today')
else:
print('cool today')
```

16. Regarding proper indentation,
(a) Explain the change in indentation needed in order for the following code to be syntactically correct. **(b)** Indicate other changes in the indentation of the code that is not strictly needed, but would make the

code more readable.

```
if level ,5 1:
print('Value is well within range')
print('Recheck in one year')

elif level ,5 2:
print('Value is within range')
print('Recheck within one month')

elif level ,5 3:
print('Value is slightly high)
print('Recheck in one week')

elif level ,5 4:
print('Value abnormally high')
print('Shut down system immediately')
```

Section 3.4
17. Write a program segment that uses a while loop to add up all the even numbers between 100 and 200, inclusive.
18. The following while loop is meant to multiply a series of integers input by the user, until a sentinel value of 0 is entered. Indicate any errors in the code given.

```
product 5 1
num 5 input('Enter first number: ') while num ! 5 0:

num 5 input('Enter first number: ') product 5 product * num
print('product 5 ', product)
```

19. For each of the following, indicate which is a definite loop, and which is an indefinite loop. **(a)**

```
num 5 input('Enter a non-zero value: ')
while num 5 5 0:
num 5 input('Enter a non-zero value: ')
```

(b) num 5 0
while n , 10:
print 2 ** n
n 5 n 1 1

PYTHON PROGRAMMING EXERCISES

P1. Write a Python program in which the user enters either 'A', 'B', or 'C'. If 'A' is entered, the program should display the word 'Apple'; if 'B' is entered, it displays 'Banana'; and if 'C' is entered, it displays 'Coconut'. Use nested if statements for this as depicted in Figure 3-13.

P2. Repeat question P1 using an if statement with elif headers instead.

P3. Write a Python program in which a student enters the number of college credits earned. If the number of credits is greater than 90, 'Senior Status' is displayed; if greater than 60, 'Junior Status' is displayed; if greater than 30, 'Sophomore Status' is displayed; else, 'Freshman Status' is displayed.

P4. Write a program that sums a series of (positive) integers entered by the user, excluding all numbers that are greater than 100.
P5. Write a program, in which the user can enter any number of positive and negative integer values, that displays the number of positive values entered, as well as the number of negative values.
P6. Write a program containing a pair of nested while loops that displays the integer values 1–100, ten numbers per row, with the columns aligned as shown below,

1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
.
.
91 92 93 94 95 96 97 98 99 100

P7. Display the integer values 1–100 as given in question P6 using only *one* while loop.

PROGRAM MODIFICATION PROBLEMS

M1. Temperature Conversion Program: Input Error Checking
Modify the Temperature Conversion program in Figure 3-19 to perform input error checking of entered temperatures. On the Fahrenheit scale, absolute zero is 2459.67. Therefore, all valid Fahrenheit temperatures start at that value (with no upper limit). On the Celsius scale, absolute zero is 2273.15. The program should reprompt the user for any invalid entered temperatures.

M 2. Temperature Conversion Program: Addition of Kelvin Scale
Modify the Temperature Conversion program in Figure 3-19 to add an additional option of converting to and from degrees Kelvin. The formula for conversion to Kelvin (K) from Celsius (C) is K 5 C 1 273.15.

M3. Number of Days in Month Program: Input Error Checking
Modify the Number of Days in Month Program of section 3.3.4 so that the program prompts the user to re-enter any month (not in the range 1–12) or year that is an invalid value.

M4. Number of Days in Month Program: Indication of Leap Years
Modify the Number of Days in Month program of section 3.3.4 so that the program displays, in addition to the number of days in the month, that the year is a leap year or not as shown below.

Enter the month (1-12): 2
Please enter the year (e.g., 2010): 2000 There are 29 days in the month (a leap year)

M5. Oil Change Notification Program: Number of Miles before Change
Modify the Oil Change Notification program in Figure 3-21 so that the program displays the number of miles left before the next oil change, or the number of miles overdue for an oil change, as appropriate.

M6. Coin Change Exercise Program: Addition of Half-Dollar Coins
Modify the Coin Change Exercise program in section 3.4.6 to allow for the use of half-dollar coins. Make all necessary changes in the program.

M7. Coin Change Exercise Program: Raising the Challenge
Modify the Coin Change Exercise program in section 3.4.6 so that the least possible number of coins must be entered. For example, the least number of coins that total to 43 cents is 6 (one quarter, one dime, one nickel, and three pennies).

M8. Calendar Month Program: Indication of Leap Year
Modify the final version of the Calendar Month program in section 3.5 so that for leap years, the month heading is displayed as in the following,

```
June   1984 (leap year)

                        1   2

   3   4   5   6   7   8   9

  10  11  12  13  14  15  16

  17  18  19  20  21  22  23

  24  25  26  27  28  29  30
```

M9. Calendar Month Program: User Entry of Month Name
Modify the final version of the Calendar Month program to allow the user to enter a month's name (e.g., 'January') rather than a number (e.g., 1). Make all appropriate changes in the program as a result of this change.
M10. Calendar Month Program: Day of the Week Headings

Modify the final version of the Calendar Month program in section 3.5 so that there is day heading for each of the columns as shown below.

```
January   1800

  Su  Mo  Tu  We  Th  Fr  Sa
              1   2   3   4
   5   6   7   8   9  10  11
  12  13  14  15  16  17  18
  19  20  21  22  23  24  25
  26  27  28  29  30  31
```

M11. Sage Program Modification
Following is the output of an "all knowing" Sage program that replies with random responses to questions posed by the user. The responses generated have no meaningful connection to the questions asked.

```
What is your question?:
When will I finally finish the book?

You ask me "When will I finally finish the book?" . . .
You know the answer to that already, don't you?

Do you have another question? (y/n): y
What is your question?:
Do you really know what I am asking?

You ask me "Do you really know what I am asking?" . . .
I would focus my thoughts on something else.

Do you have another question? (y/n): y
What is your question?:
Can't you answer my questions directly?

You ask me "Can't you answer my questions directly?" . . .
The probabilities are in your favor.

Do you have another question? (y/n): y
What is your question?:
So, what's the difference between an orange?

You ask me "So, what's the difference between an orange?" . . .
It is close to certainty.

Do you have another question? (y/n): y
What is your question?:
I think you are bogus, aren't you?

You ask me "I think you are bogus, aren't you?" . . .
Someone you would not expect can be most helpful about this.

Do you have another question? (y/n): n
Goodbye ... hope I was of some help!
>>>
```

Program Development Problems 123

In the 1960s, a program called Eliza was developed that was able to behave as a psychotherapist. It did not really understand anything, it only looked for certain words to turn the patient's comments or questions back to the patient. For example, if a patient said, "My mom drives me crazy," it might reply with "Tell me more about your mom." Modify this program so that it appears to have understanding by similar means of word recognition as used in the Eliza program. Specifically, incorporate a set of "trigger" words that, if found, causes a specific response to be given. For example, if the word "I" appears in the question (for example, "Will I ever be rich?" or "Am I always going to be

happy?”), the response may be “You are in charge of your own destiny.” If the word “new” appears in the question (for example, “Will I find a new boyfriend soon?” or “Will I find a new life?,” the response may be “Changes are up to you and the unpredictable events in life.”

Be creative. In order to determine if a given word (or phrase) appears in a given question, make use of the in membership operator.

```
# Fortune Teller Program
import random

have_question = 'y'

while have_question == 'y':
    question = input('What is your question?:\n')
    print('\nYou ask me', '"' + question + '" . . .')

    rand_num = random.randint(1,8)

    if rand_num == 1:
        print ('The probabilities are in your favor')
    elif rand_num == 2:
        print ("I wouldn't make any definite plans")
    elif rand_num == 3:
        print ('The outlook is dim')
    elif rand_num == 4:
        print ('I would focus my thoughts on something else')
    elif rand_num == 5:
        print ('You are the only one that can answer that!')
    elif rand_num == 6:
        print ("You know the answer to that already, don't you?")
    elif rand_num == 7:
        print ('Someone unexpected can be most helpful with this')
    elif rand_num == 8:
        print ('It is close to certainty')

    have_question = input('\nDo you have another question? (y/n): ')

print('Goodbye ... hope I was of some help!')
```

PROGRAM DEVELOPMENT PROBLEMS

D 1. Metric Conversion
Develop and test a Python program that converts pounds to grams, inches to centimeters, and kilometers to miles. The program should allow conversions both ways.

D2. Leap Years to Come
Develop and test a Python program that displays future leap years, starting with

the first occurring leap year from the current year, until a final year entered by the user. (HINT: Module datetime used in the Age in Seconds Program of Chapter 2 will be needed here.)

D3. The First-Time Home Buyer Tax Credit
Develop and test a Python program that determines if an individual qualifies for a government First-Time Home Buyer Tax Credit of $8,000. The credit was only available to those that (a) bought a house that cost less than $800,000, (b) had a combined income of under $225,000 and (c) had not owned a primary residence in the last three years.

D4. Home Loan Amortization
Develop and test a Python program that calculates the monthly mortgage payments for a given loan amount, term (number of years) and range of interest rates from 3% to 18%. The fundamental formula for determining this is A/D, where A is the original loan amount, and D is the discount factor. The discount factor is calculated as,

D 5 ((1 1 r) 2 1) / r(1 1 r)

where is the number of total payments (12 times the number of years of the loan) and r is the interest rate, expressed in decimal form (e.g., .05), divided by 12. A monthly payment table should be generated as shown below,

Loan Amount: $350,000 Term: 30 years Interest Rate Monthly Payment 3% 1475.61 4% 1670.95 5% 1878.88 6% 2098.43 . .
. .
18% 5274.80

Check your results with an online mortgage calculator.
D5. Life Signs
Develop and test a program that determines how many breaths and how many heartbeats a person has had
in their life. The average respiration (breath) rate of people varies with age. Use the breath rates given
below for use in your program,

Breaths per Minute
Infant 25–60 1–4 years 20–30
5–14 years 15–25

15–18 years 11–23

For heart rate, use an average of 67.5 beats per second.

Lists CHAPTER 4

In this chapter, we look at a means of structuring and accessing a collection of data. In particular, we look at a way of organizing data in a linear sequence, generally referred to as a list.

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Explain what a list is in programming
♦ Describe the typical operations performed on lists
♦ Explain what is meant by list traversal
♦ Effectively create and use lists in Python
♦ Explain the difference between lists and tuples in Python
♦ Explain what a sequence is in Python
♦ Describe the sequence operations common to lists, tuples, and strings in Python ♦ Effectively use nested lists and tuples in Python
♦ Effectively iterate over lists (sequences) in Python
♦ Effectively use for statements for iterative control in Python
♦ Use the range function in Python
♦ Explain how list representation relates to list assignment in Python ♦ Effectively use list comprehensions in Python
♦ Write Python programs using sequences

CHAPTER CONTENTS
Motivation
Fundamental Concepts
4.1 List Structures
4.2 Lists (Sequences) in Python

4.3 Iterating Over Lists (Sequences) in Python
4.4 More on Python Lists
Computational Problem Solving
4.5 Calendar Year Program

MOTIVATION

The way that data is organized has a significant impact on how effec tively it can be used. One of the most obvious and useful ways to organize data is as a list. We use lists in our everyday lives—we make shopping lists, to-do lists, and mental checklists. Various forms of lists are provided by programming languages, differing in the ele ments they can store (mixed type?), their size (variable size?), whether they can be altered (mutable?), and the operations that can be per formed on them (see Figure 4-1).

Lists also occur in nature. Our DNA is essentially a long list of molecules in the form of a double helix, found in the nucleus of all human cells and all living organisms. Its purpose is also to store information—specifically, the instructions that are used to construct all other cells in the body—that we call *genes*. Given the 2.85 billion nucleotides that make up the human genome, determining their sequencing (and thus understanding our genetic makeup) is fundamentally a computational problem.

In this chapter, we look at the use of lists and other sequences in Python.

List Characteristics	Elements
Element Type	All elements of the same type
	Elements of different types
Length	Fixed length
	Varying length
Modifiability	Mutable (alterable)
	Immutable (unalterable)
Common Operations	Determine if a list is empty
	Determine the length of a list
	Access (retrieve) elements of a list
	Insert elements into a list
	Replace elements of a list
	Delete elements of a list
	Append elements to (the end of) a list

FIGURE 4-1 List Properties and Common Operations

FUNDAMENTAL CONCEPTS

4.1 List Structures

In this section we introduce the use of lists in programming. The concept of a list is similar to our everyday notion of a list. We read off (access) items on our to-do list, add items, cross off (delete) items, and so forth. We look at the use of lists next.

4.1.1 What Is a List?

A **list** is a *linear data structure*, meaning that its elements have a linear ordering. That is, there is a first element, a second element, and so on. Figure 4-2 depicts a list storing the average temperature for each day of a given week, in which each item in the list is identified by its *index value*.

The location at index 0 stores the temperature for Sunday, the location at index 1 stores the temperature for Monday, and so on. It is customary in programming languages to begin numbering sequences of items with an index value of 0 rather than 1. This is referred to as *zero-based indexing*. This is important to keep in mind to avoid any “off by one” errors in programs, as we shall see. We next look at some common operations performed on lists.

0:	68.8
1:	70.2
2:	67.2
3:	71.8
4:	73.2
5:	75.6
6:	74.0

FIGURE 4-2
Indexed Data
Structure daily_temperatures

A **list** is a linear data structure, thus its elements have a linear ordering.

4.1.2 Common List Operations

Operations commonly performed on lists include retrieve, update, insert, delete

(remove) and append. Figure 4-3 depicts these operations on a list of integers.

FIGURE 4-3 Common List Operations

The operation depicted in (a) retrieves elements of a list by index value. Thus, the value 50 is retrieved at index 4 (the fifth item in the list). The replace operation in (b) updates the current value at index 4, 50, with 55. The insert operation in (c) inserts the new value 25 at index 2, thus shifting down all elements below that point and lengthening the list by one. In (d), the remove operation deletes the element at index 6, thus shifting up all elements below that point and shortening the list by one. Finally, the append operation in (e) adds a new value, 80, to the end of the list. In the following sections we will see how these operations are accomplished in Python. First, we look at what is called *list traversal*, a way of accessing each of the elements of a given list.

Operations commonly performed on lists include retrieve, update, insert, remove, and append.

4.1.3 List Traversal

A **list traversal** is a means of accessing, one-by-one, the elements of a list. For example, to add up all the elements in a list of integers, each element can be accessed one-by-one, starting with the first, and ending with the last element. Similarly, the list could be traversed starting with the last element and ending with the first. To find a particular value in a list also requires traversal. We depict the tasks of summing and searching a list in Figure 4-4.

FIGURE 4-4 List Traversal

A **list traversal** is a means of accessing, one-by-one, the elements of a list.

Self-Test Questions

1. What would be the range of index values for a list of 10 elements?
(a) 0–9 **(b)** 0–10 **(c)** 1–10

2. Which one of the following is NOT a common operation on lists?
(a) access **(b)** replace **(c)** interleave **(d)** append **(e)** insert **(f)** delete

3. Which of the following would be the resulting list after inserting the value 50 at index 2?

0: 35
1: 15
2: 45

3: 28

(a) 0: 35 (b) 0: 35 (c) 0: 50
1: 50 1: 15 1: 35
2: 15 2: 50 2: 15
3: 45 3: 45 3: 45
4: 28 4: 28 4: 28

ANSWERS: 1. a, 2. c, 3. b

4.2 Lists (Sequences) in Python

Next, we look at lists (and other sequence types) in Python.

4.2.1 Python List Type

A **list** in Python is a mutable, linear data structure of variable length, allowing mixed-type elements. *Mutable* means that the contents of the list may be altered. Lists in Python use zerobased indexing. Thus, all lists have index values 0 ... n-1, where n is the number of elements in the list. Lists are denoted by a comma-separated list of elements within square brackets as shown below,

[1, 2, 3] ['one', 'two', 'three'] ['apples', 50, True]

An **empty list** is denoted by an empty pair of square brackets, []. (We shall later see the usefulness of the empty list.) Elements of a list are accessed by using an index value within square brackets,

lst 5 [1, 2, 3] lst[0] → 1 access of first element lst[1] → 2 access of second element lst[2] → 3 access of third element

Thus, for example, the following prints the first element of list lst,
print(lst[0])
The elements in list lst can be summed as follows,
sum 5 lst[0] 1 lst[1] 1 lst[2]

For longer lists, we would want to have a more concise way of traversing the elements. We discuss this below. Elements of a list can be updated (replaced) or deleted (removed) as follows (for lst 5 [1, 2, 3]),

lst[2] 5 4 [1, 2, 4] replacement of 3 with 4 at index 2 del lst[2] [1, 2] removal of 4 at index 2
Methods insert and append also provide a means of altering a list,

lst.insert(1, 3) [1, 3, 2] insertion of 3 at index 1 lst.append(4) [1, 3, 2, 4] appending of 4 to end of list
In addition, methods sort and reverse reorder the elements of a given list. These list modifying operations are summarized in Figure 4-5.

Operation	fruit = ['banana', 'apple, 'cherry']	
Replace	fruit[2] = 'coconut'	['banana', 'apple', 'coconut']
Delete	del fruit[1]	['banana', 'cherry']
Insert	fruit.insert(2, 'pear')	['banana', 'apple', 'pear', 'cherry']
Append	fruit.append('peach')	['banana', 'apple', 'cherry', 'peach']
Sort	fruit.sort()	['apple', 'banana', 'cherry']
Reverse	fruit.reverse()	['cherry', 'banana', 'apple']

FIGURE 4-5 List Modification Operations in Python

Methods , and the associated *dot notation* used, are fully explained in Chapter 6 on Objects and Their Use. We only mention methods here for the sake of completeness in covering the topic of list operations.

LET'S TRY IT From the Python Shell, enter the following and observe the results.

... lst 5 [10, 20, 30] ... del lst[2] ... lst ... lst
??? ???

... lst[0] ... lst.insert(1, 15) ??? ... lst
???

... lst[0] 5 5 ... lst.append(40) ... lst ... lst
??? ???

A **list** in Python is a mutable linear data structure, denoted by a comma-separated list of elements within square brackets, allowing mixed-type elements.
4.2.2 Tuples

A **tuple** is an *immutable* linear data structure. Thus, in contrast to lists, once a tuple is defined, it cannot be altered. Otherwise, tuples and lists are essentially the same. To distinguish tuples from lists, tuples are denoted by parentheses instead of square brackets as given below,

nums 5 (10, 20, 30)
student 5 ('John Smith', 48, 'Computer Science', 3.42)

Another difference between tuples and lists is that *tuples of one element must include a comma following the element.* Otherwise, the parenthesized element will not be made into a tuple, as shown below,

CORRECT WRONG ... (1,) ... (1)
(1) 1

An **empty tuple** is represented by a set of empty parentheses, (). (We shall later see the usefulness of the empty tuple.) The elements of tuples are accessed the same as lists, with square brackets,

... nums[0] ... student[0]
10 'John Smith'

Any attempt to alter a tuple is invalid. Thus, delete, update, insert, and append operations are not defined on tuples. For now, we can consider using tuples when the information to represent should not be altered. We will see additional uses of tuples in the coming chapters.

LET'S TRY IT From the Python Shell, enter the following and observe the results.

... t 5 (10, 20, 30) ... t.insert(1, 15) ... t[0] ... ???
??? ???

... del t[2] ... t.append(40) ??? ???
A **tuple** in Python is an *immutable* linear data structure, denoted by a comma-separated list of elements within parentheses, allowing mixed-type elements.
4.2.3 Sequences

A **sequence** in Python is a linearly ordered set of elements accessed by an index number. Lists, tuples, and strings are all sequences. Strings, like tuples, are

immutable *;* therefore, they cannot be altered. We give sequence operations common to strings, lists, and tuples in Figure 4-6.

For any sequence s, len(s)gives its length, and s[k] retrieves the element at index k. The slice operation, s[index1:index2], returns a subsequence of a sequence, starting with the first index location up to *but not including* the second. The s[index:] form of the slice operation returns a string containing all the list elements starting from the given index location to the end of the sequence. The count method returns how many instances of a given value occur within a sequence, and the find method returns the index location of the *first occurrence* of a specific item, returning 21 if not found. For determining only if a given value occurs within a

Operation		**String** s = 'hello' w = '!'	**Tuple** s = (1,2,3,4) w = (5,6)	**List** s = [1,2,3,4] w = [5,6]
Length	len(s)	5	4	4
Select	s[0]	'h'	1	1
Slice	s[1:4] s[1:]	'ell' 'ello'	(2, 3, 4) (2, 3 ,4)	[2 ,3, 4] [2, 3, 4]
Count	s.count('e') s.count(4)	1 *error*	0 1	0 1
Index	s.index('e') s.index(3)	1 --	-- 2	-- 2
Membership	'h' in s	True	False	False
Concatenation	s + w	'hello!'	(1, 2, 3, 4, 5, 6)	[1, 2, 3, 4, 5, 6]
Minimum Value	min(s)	'e'	1	1
Maximum Value	max(s)	'o'	4	4
Sum	sum(s)	*error*	10	10

FIGURE 4-6 Sequence Operations in Python

sequence, without needing to know where, the in operator (introduced in Chapter 3) can be used instead.

The 1 operator is used to denote concatenation. Since the plus sign also denotes addition, Python determines which operation to perform based on the operand types. Thus the plus sign, 1, is referred to as an **overloaded operator**. If both operands are numeric types, addition is performed. If both operands are

sequence types, concatenation is performed. (If a mix of numeric and sequence operands is used, an “unsupported operand type(s) for 1” error message will occur.) Operations min/max return the smallest/largest value of a sequence, and sum returns the sum of all the elements (when of numeric type). Finally, the comparison operator, 55, returns True if the two sequences are the same length, and their corresponding elements are equal to each other.

LET’S TRY IT From the Python Shell, enter the following and observe the results.

... s 5 'coconut' ... s 5 (10, 30, 20, 10) ... s 5 [10, 30, 20, 10] ... s[4:7] ... s[1:3] ... s[1:3] ??? ??? ???

... s.count('o') ... s.count(10) ... s.count(10) ??? ??? ???
... s.index('o') ... s.index(10) ... s.index(10) ??? ??? ???
... s 1 ' juice' ... s 1 (40, 50) ... s 1 (40, 50) ??? ??? ???
In Python, a **sequence** is a linearly ordered set of elements accessed by index value. Lists, tuples, and strings are sequence types in Python.
4.2.4 Nested Lists

Lists and tuples can contain elements of any type, including other sequences. Thus, lists and tuples can be nested to create arbitrarily complex data structures. Below is a list of exam grades for each student in a given class,

class_grades 5 [[85, 91, 89], [78, 81, 86], [62, 75, 77], ...]

In this list, for example, class_grades[0] equals [85, 91, 89], and class_grades[1] equals [78, 81, 86]. Thus, the following would access the first exam grade of the first student in the list,

student1_grades 5 class_grades[0] student1_exam1 5 student1_grades[0]
However, there is no need for intermediate variables student1_grades and student1_ exam1. The exam grade can be directly accessed as follows,
class_grades[0][0] → [85, 91, 89][0] → 85
To calculate the class average on the first exam, a while loop can be constructed that iterates over the first grade of each student’s list of grades,

sum 5 0
k 5 0
while k , len(class_grades):

sum 5 sum 1 class_grades[k][0]
k 5 k 1 1
average_exam1 5 sum / float(len(class_grades)) If we wanted to produce a new list containing the exam average for each student in the class, we could do the following,

exam_avgs 5 []
k 5 0
while k , len(class_grades):

avg 5 (class_grades[k][0] 1 class_grades[k][1] 1 \ class_grades[k][2]) / 3.0
exam_avgs.append(avg)
k 5 k 1 1

Each time through the loop, the average of the exam grades for a student is computed and appended to list exam_avgs. When the loop terminates, exam_avgs will contain the corresponding exam average for each student in the class.
LET'S TRY IT

From the Python Shell, enter the following and observe the results.

... lst 5 [[1, 2 ,3], [4, 5, 6], [7, 8, 9]]
... lst[0] ... lst[1] ??? ??? ... lst[0][1] ... lst[1][1] ??? ???

Lists and tuples can be nested within each other to construct arbitrarily complex data structures.
4.2.5 Let's Apply It—A Chinese Zodiac Program
The following program (Figure 4-8) determines the animal and associated characteristics from the Chinese Zodiac for a given year of birth. This program utilizes the following programming features:
➤ tuples ➤ datetime module
Example execution of the program is given in Figure 4-7.

```
This program will display your Chinese zodiac sign and associated
personal characteristics.

Enter your year of birth (yyyy): 1984
Your Chinese zodiac sign is the Rat

Your personal characteristics ...
Forthright, industrious, sensitive, intellectual, sociable

Would you like to enter another year? (y/n): y
Enter your year of birth (yyyy): 1986
Your Chinese zodiac sign is the Tiger

Your personal characteristics ...
Unpredictable, rebellious, passionate, daring, impulsive

Would you like to enter another year? (y/n): n
>>>
```

FIGURE 4-7 Execution of the Chinese Zodiac Program

Line 3 imports the datetime module. It provides the current year (**line 31**), used to check for invalid years of birth (only years between 1900 and the current year are considered valid). **Lines 9–24** perform the initialization for the program. The variables on lines **9–20** are assigned the characteristics of each animal. The set of characteristics is represented as a tuple

```
1  # Chinese Zodiac Program
2
3  import datetime
4
5  # init
6  zodiac_animals = ('Rat', 'Ox', 'Tiger', 'Rabbit', 'Dragon', 'Snake', 'Horse',
7                    'Goat', 'Monkey', 'Rooster', 'Dog', 'Pig')
8
9  rat = 'Forthright, industrious, sensitive, intellectual, sociable'
10 ox = 'Dependable, methodical, modest, born leader, patient'
11 tiger = 'Unpredictable, rebellious, passionate, daring, impulsive'
12 rabbit = 'Good friend, kind, soft-spoken, cautious, artistic'
13 dragon = 'Strong, self-assured, proud, decisive, loyal'
14 snake = 'Deep thinker, creative, responsible, calm, purposeful'
15 horse = 'Cheerful, quick-witted, perceptive, talkative, open-minded'
16 goat = 'Sincere, sympathetic, shy, generous, mothering'
17 monkey = 'Motivator, inquisitive, flexible, innovative, problem solver'
18 rooster = 'Organized, self-assured, decisive, perfectionist, zealous'
19 dog = 'Honest, unpretentious, idealistic, moralistic, easy going'
20 pig = 'Peace-loving, hard-working, trusting, understanding, thoughtful'
21
22 characteristics = (rat, ox, tiger, rabbit, dragon, snake, horse, goat, monkey,
23                    rooster, dog, pig)
24 terminate = False
25
26 # program greeting
27 print('This program will display your Chinese zodiac sign and associated')
28 print('personal characteristics.\n')
29
30 # get current year from module datetime
31 current_yr = datetime.date.today().year
32
33 while not terminate:
34
35     # get year of birth
36     birth_year = int(input('Enter your year of birth (yyyy): '))
37
38     while birth_year < 1900 or birth_year > current_yr:
39         print('Invalid year. Please re-enter\n')
40         birth_year = int(input('Enter your year of birth (yyyy): '))
41
42     # output results
43     cycle_num = (birth_year - 1900) % 12
44
45     print('Your Chinese zodiac sign is the', zodiac_animals[cycle_num],'\n')
46     print('Your personal characteristics ...')
47     print(characteristics[cycle_num])
48
49     # continue?
50     response = input('\nWould you like to enter another year? (y/n): ')
51
52     while response != 'y' and response != 'n':
53         response = input("Please enter 'y' or 'n': ")
54
55     if response == 'n':
56         terminate = True
```

FIGURE 4-8 Chinese Zodiac Program

(**line 22**), and not a list type, since the information is not meant to be altered. It

associates each set of characteristics with the corresponding year of the twelve-year cycle of the zodiac based on their position in the tuple. (We could have defined characteristics to contain each of the twelve string descriptions, without the use of variables rat, ox, and so on. It was written this way for the sake of readability.) Variable terminate, initialized to False, is a Boolean flag used to quit the program once set to True (in response to the user being asked to continue with another month or not at **line 50**). **Lines 27–28** display the program greeting.

Lines 33–56 comprise the main loop of the program. The while loop at **line 38** ensures that the entered year is valid. On **line 43**, the cycle_num for the individual is assigned a value between 0–11, based on their year of birth. Since the year 1900 was the year of the rat in the Chinese Zodiac, the value of cycle_num is (birth_year — 1900) % 12. **Lines 45–47** then use the cycle_num as an index into tuple zodiac_animals (to get the animal for that birth year) and tuple characteristics (to get the associated personal characteristics) to display the results.

Self-Test Questions

1. Which of the following sequence types is a mutable type?
(a) strings **(b)** lists **(c)** tuples

2. Which of the following is true?
(a) Lists and tuples are denoted by the use of square brackets.
(b) Lists are denoted by use of square brackets and tuples are denoted by the use of

parentheses.
(c) Lists are denoted by use of parentheses and tuples are denoted by the use of square
brackets.

3. Lists and tuples must each contain at least one element. (TRUE/FALSE)

4. For lst 5 [4, 2, 9, 1], what is the result of the following operation, lst.insert(2, 3)?
(a)[4, 2, 3, 9, 1] **(b)**[4, 3 ,2, 9, 1] **(c)** [4, 2, 9, 2, 1]

5. Which of the following is the correct way to denote a tuple of one element?
(a) [6] **(b)** (6) **(c)** [6,] **(d)** (6,)

6. Which of the following set of operations can be applied to any sequence?
(a)len(s), s[i], s 1w (concatenation)
(b)max(s), s[i], sum(s)
(c)len(s), s[i], s.sort()

ANSWERS: 1. (b), 2. (b), 3. False, 4. (a), 5. (d), 6. (a)

4.3 Iterating Over Lists (Sequences) in Python

Python's for statement provides a convenient means of iterating over lists (and other sequences). In this section, we look at both for loops and while loops for list iteration.

4.3.1 For Loops

A **for statement** is an iterative control statement that iterates once for each element in a specified sequence of elements. Thus, for loops are used to construct definite loops. For example, a for loop is given in Figure 4-9 that prints out the values of a specific list of integers.

for statement	Example use

`for k in sequence:`
`  suite`
	`nums = [10, 20, 30, 40, 50, 60]`
	
	`for k in nums:`
	`  print(k)`

FIGURE 4-9 The for Statement in Python

V ariable k is referred to as a **loop variable**. Since there are six elements in the provided list, the for loop iterates exactly six times. To contrast the use of for loops and while loops for list iteration, the same iteration is provided as a while loop below,

k 5 0
while k , len(nums):
print(nums[k])
k 5 k 1 1

In the while loop version, loop variable k must be initialized to 0 and incremented by 1 each time through the loop. In the for loop version, loop variable k *automatically* iterates over the provided sequence of values.

The for statement can be applied to all sequence types, including strings. Thus, iteration over a string can be done as follows (which prints each letter on a separate line).
for ch in 'Hello':
print(ch)
Next we look at the use of the built-in range function with for loops.
LET’S TRY IT From the Python Shell, enter the following and observe the results.

... for k in [4, 2 ,3, 1]: ... for k in ['Apple', 'Banana', 'Pear']: print(k) print(k)
??? ???

... for k in (4, 2, 3, 1): ... for k in 'Apple':
print(k) print(k)
??? ???

A **for statement** is an iterative control statement that iterates once for each element in a specified sequence of elements.
4.3.2 The Built-in **range** Function
Python provides a built-in **range function** that can be used for generating a sequence of integers that a for loop can iterate over, as shown below.

sum 5 0
for k in range(1, 11):
sum 5 sum 1 k

The values in the generated sequence include the starting value, up to *but not including* the ending value. For example, range(1, 11) generates the sequence [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Thus, this for loop adds up the integer values 1–10.

The range function is convenient when long sequences of integers are needed. Actually, range does not create a sequence of integers. It creates a *generator function* able to produce each next item of the sequence when needed. This saves memory, especially for long lists. Therefore, typing range(0, 9) in the Python shell does not produce a list as expected—it simply “echoes out” the call to range.

By default, the range function generates a sequence of consecutive integers. A “step” value can be provided, however. For example, range(0, 11, 2) produces the sequence [0, 2, 4, 6, 8, 10], with a step value of 2. A sequence can also be

generated “backwards” when given a negative step value. For example, range(10, 0, 2 1) produces the sequence [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]. Note that since the generated sequence always begins with the provided starting value, “up to” but not including the final value, the final value here is 0, and not 1.

LET’S TRY IT From the Python Shell, enter the following and observe the results.

... for k in range(0, 11): ... for k in range(2, 102, 2): print(k) print(k)
??? ???

... for k in range[0, 11]: ... for k in range(10, 21, 22): print(k) print(k)
??? ???

Python provides a built-in **range** function that can be used for generating a sequence of integers that a for loop can iterate over.
4.3.3 Iterating Over List Elements vs. List Index Values

When the elements of a list need to be accessed, but not altered, a loop variable that iterates over each list element is an appropriate approach. However, there are times when the loop variable must iterate over the *index values* of a list instead. A comparison of the two approaches is shown in Figure 4-10.

Loop variable iterating over the elements of a sequence	Loop variable iterating over the index values of a sequence

`nums = [10, 20, 30, 40, 50, 60]`
 	
`for k in nums:`
`  sum = sum + k`
	`nums = [10, 20, 30, 40, 50, 60]`
	
	`for k in range(len(nums)):`
	`  sum = sum + nums[k]`

FIGURE 4-10 Iterating Over the Elements vs. the Index Values of a Given Sequence
Suppose the average of a list of class grades named grades needs to be computed. In this case, a for loop can be constructed to iterate over the grades,
for k in grades:
sum 5 sum 1 k
print('Class average is', sum/len(grades))

However, suppose that the instructor made a mistake in grading, and a point

needed to be added to each student’s grade? In order to accomplish this, the index value (the location) of each element must be used to update each grade value. Thus, the loop variable of the for loop must iterate over the index values of the list,

for k in range(len(grades)): grades[k] 5 grades[k] 1 1

In such cases, the loop variable k is also functioning as an *index variable*. An **index variable** is a variable whose *changing value is used to access elements of an indexed data structure*. Note that the range function may be given only one argument. In that case, the starting value of the range defaults to 0. Thus, range(len(grades)) is equivalent to range(0,len(grades)).

LET’S TRY IT From the Python Shell, enter the following and observe the results. ... nums 5 [10, 20, 30]

... for k in range(len(nums)): print(nums[k])
???
... for k in range(len(nums)21, 21, 21): print(nums[k])
???

An **index variable** is a variable whose changing value is used to access elements of an indexed data structure.
4.3.4 While Loops and Lists (Sequences)

There are situations in which a sequence is to be traversed while a given condition is true. In such cases, a while loop is the appropriate control structure. (Another approach for the partial traversal of a sequence is by use of a for loop containing break statements. We avoid the use of break statements in this text, favoring the more structured while loop approach.)

Let’s say that we need to determine whether the value 40 occurs in list nums (equal to [10, 20, 30]). In this case, once the value is found, the traversal of the list is terminated. An example of this is given in Figure 4-11.

V ariable k is initialized to 0,and used as an index variable. Thus, the first time through the loop, k is 0, and nums[0] (with the value 10) is compared to item_to_find. Since they are not equal, the second clause of the if statement is executed, incrementing k to 1. The loop continues until either the item is found, or the complete list has been traversed. The final if statement determines

```
k = 0
item_to_find = 40
found_item = False

while k < len(nums) and not found_item:
    if nums[k] == item_to_find:
        found_item = True
    else:
        k = k + 1

if found_item:
    print('item found')
else:
    print('item not found')
```

FIGURE 4-11 List Search in Python

which of the two possibilities for ending the loop occurred, displaying either 'item found' or 'item not found'. Finally, note that the correct loop condition is k , len(nums), and not k ,5 len(nums). Otherwise, an “index out of range” error would result.

LET’S TRY IT
Enter and execute the following Python code and observe the results.

k 5 0
sum 5 0
nums 5 range(100)

while k , len(nums) and sum , 100:
sum 5 sum 1 nums[k]
k 5 k 1 1

print('The first', k, 'integers sum to 100 or greater')
For situations in which a sequence is to be traversed while a given condition is true, a while loop is the appropriate control structure to use.
4.3.5 Let’s Apply It—Password Encryption/Decryption Program

The following program (Figure 4-13) allows a user to encrypt and decrypt passwords containing uppercase/lowercase characters, digits, and special characters. This program utilizes the following programming features:

➤ for loop ➤ nested sequences (tuples)

Example program execution is given in Figure 4-12.

Lines 4–9 perform the initialization needed for the program. Variable password_out is used to hold the encrypted or decrypted output of the program. Since the output string is created by appending to it each translated character one at a time, it is initialized to the empty string.

Program Execution …

```
This program will encrypt and decrypt user passwords

Enter (e) to encrypt a password, and (d) to decrypt: e
Enter password: Pizza2Day!
Your encrypted password is: Njaam2Fmc!
>>>
```

Program Execution …

```
This program will encrypt and decrypt user passwords

Enter (e) to encrypt a password, and (d) to decrypt: d
Enter password: Njaam2Fmc!
Your decrypted password is: Pizza2Day!
```

FIGURE 4-12 Execution of Password Encryption/Decryption Program

Variable encryption_key holds the tuple (of tuples) used to encrypt/decrypt passwords. This tuple contains as elements tuples of length two, encryption_key 5 (('a', 'm'), ('b', 'h'), etc.

The first tuple, ('a', 'm'), for example, is used to encode the letter 'a'. Thus, when encrypting a given file, each occurrence of 'a' is replaced by the letter 'm'. When decrypting, the reverse is done—all occurrences of letter 'm' are replaced by the letter 'a'.

Line 12 contains the program greeting. **Line 15** inputs from the user whether they wish to encrypt or decrypt a password. Based on the response, variable encrypting is set to either True or False (**line 20**).

The program section in **lines 26–47** performs the encryption and decryption. If variable encrypting is equal to True, then from_index is set to 0 and to_index is set to 1, causing the “direction” of the substitution of letters to go from the first in the pair to the second ('a' replaced by 'm'). When encrypting is False (and thus

decryption should be performed), the direction of the substitution is from the second of the pair to the first ('m' replaced by 'a').

V ariable case_changer (**line 33**) is set to the difference between the encoding of the lowercase and the uppercase letters (recall that the encoding of the lowercase letters is greater than that of the uppercase letters). The for loop at **line 38** performs the iteration over the pairs of letters in the encryption key. The first time through the loop, t 5('a', 'm'). Thus, t[from_index] and t[to_index] refer to each of the characters in the pair. Since all characters in the encryption key are in lowercase, when uppercase letters are found in the password, they are converted to lowercase by use of variable case_changer (**line 43**) before being compared to the (lowercase) letters in the encryption key. This works because the character encoding of all lowercase letters is greater than the corresponding uppercase version,

... ord('A') ... ord('a') ... ord('a') 2 ord('A') 65 97 32
A similar approach is used for converting from lowercase back to uppercase. Finally, on **lines 50–53**, the encrypted and decrypted versions of the password are displayed to the user.

```
1  # Password Encryption/Decryption Program
2
3  # init
4  password_out = ''
5  case_changer = ord('a') - ord('A')
6  encryption_key = (('a','m'), ('b','h'), ('c','t'), ('d','f'), ('e','g'),
7    ('f','k'), ('g','b'), ('h','p'), ('i','j'), ('j','w'), ('k','e'),('l','r'),
8    ('m','q'), ('n','s'), ('o','l'), ('p','n'), ('q','i'), ('r','u'), ('s','o'),
9    ('t','x'), ('u','z'), ('v','y'), ('w','v'), ('x','d'), ('y','c'), ('z','a'))
10
11 # program greeting
12 print('This program will encrypt and decrypt user passwords\n')
13
14 # get selection (encrypt/decrypt)
15 which = input('Enter (e) to encrypt a password, and (d) to decrypt: ')
16
17 while which != 'e' and which != 'd':
18     which = input("\nINVALID - Enter 'e' to encrypt, 'd' to decrypt: ")
19
20 encrypting = (which == 'e')  # assigns True or False
21
22 # get password
23 password_in = input('Enter password: ')
24
25 # perform encryption / decryption
26 if encrypting:
27     from_index = 0
28     to_index = 1
29 else:
30     from_index = 1
31     to_index = 0
32
33 case_changer = ord('a') - ord('A')
34
35 for ch in password_in:
36     letter_found = False
37
38     for t in encryption_key:
39         if ('a' <= ch and ch <= 'z') and ch == t[from_index]:
40             password_out = password_out + t[to_index]
41             letter_found = True
42         elif ('A' <= ch and ch <= 'Z') and chr(ord(ch) + 32) == t[from_index]:
43             password_out = password_out + chr(ord(t[to_index]) - case_changer)
44             letter_found = True
45
46     if not letter_found:
47          password_out = password_out + ch
48
49 # output
50 if encrypting:
51     print('Your encrypted password is:', password_out)
52 else:
53     print('Your decrypted password is:', password_out)
```

FIGURE 4-13 Password Encryption/Decryption Program

The substitution occurs in the nested for loops in **lines 35–47**. The outer for loop iterates variable ch over each character in the entered password (to be encrypted or decrypted). The first step of the outer for loop is to initialize letter_found to

False. This variable is used to indicate if each character is a (uppercase or lowercase) letter. If so, it is replaced by its corresponding encoding character. If not, it must be a digit or special character, and thus appended as is (**line 47**). The code on **lines 39–41** and **lines 42–46** is similar to each other. The only difference is that since the letters in the encryption key are all lowercase, any uppercase letters in the password need to be converted to lowercase before being compared to the letters in the key.

Self-Test Questions **1.** For nums 5 [10,30,20,40], what does the following for loop output? for k in nums:
print(k)

(a) 10 **(b)**10 **(c)**10
20 30 30
30 20 20
40 40

2. For nums 5 [10, 30, 20, 40], what does the following for loop output? for k in range(1, 4):
print(nums[k])

(a) 10 **(b)**30 **(c)**10
30 20 30
20 40 20
40

3. For fruit 5 'strawberry', what does the following for loop output? for k in range(0, len(fruit), 2):
print(fruit[k], end='')
(a)srwer **(b)**tabry
4. For nums 5 [12, 4, 11, 23, 18, 41, 27], what is the value of k when the while loop terminates?
k 5 0
while k , len(nums) and nums[k] ! 5 18:
k 5 k 1 1
(a)3 **(b)**4 **(c)**5
ANSWERS: 1. (b), 2. (b), 3. (a), 4. (b)
4.4 More on Python Lists

In this section, we take a closer look at the assignment of lists. We also introduce

a useful and convenient means of generating lists that the range function cannot produce, called *list comprehensions*.

4.4.1 Assigning and Copying Lists

Because of the way that lists are represented in Python, when a variable is assigned to another variable holding a list, list2 5list1, each variable ends up referring to the *same instance* of the list in memory. This is depicted in Figure 4-14.

FIGURE 4-14 Assignment of Lists

This has important implications. For example, if an element of list1 is changed, then the corresponding element of list2 will change as well,

... list1 5 [10, 20, 30, 40]
... list2 5 list1
... list1[0] 5 5
... list1
[5, 20, 30, 40] change made in list1
... list2
[5, 20, 30, 40] change in list1 causes a change in list2

Knowing that variables list1 and list2 refer to the same list explains this behavior. This issue does not apply to strings and tuples, since they are immutable and therefore cannot be modified. When needed, a copy of a list can be made as given below,
list2 5 list(list1)
In this case, we get the following results,

... list1 5 [10, 20, 30, 40]
... list2 5 **list(list1)**
... list1[0] 5 5
... list1

[5, 20, 30, 40] change made in list1
... list2
[10, 20, 30, 40] change in list1 does NOT cause any change in list2

When copying lists that have sublists, another means of copying, called *deep copy*, may be needed. We will discuss this further in Chapter 6 when discussing objects in Python.

LET'S TRY IT
From the Python Shell, enter the following and observe the results.
... list1 5 ['red', 'blue', 'green'] ... list1 5 ['red', 'blue', 'green']

... list2 5 list1 ... list2 5 list(list1) ... list1[2] 5 'yellow' ... list1[2] 5 'yellow' ... list1
... list1
??? ???
... list2 ... list2
??? ???
When a variable is assigned to another variable holding a list, each variable ends up referring to the *same instance* of the list in memory.
4.4.2 List Comprehensions

The range function allows for the generation of sequences of integers in fixed increments. **List comprehensions** in Python can be used to generate more varied sequences. Example list comprehensions are given in Figure 4-15.

Example List Comprehensions	Resulting List
(a) [x**2 for x in [1, 2, 3]]	[1, 4, 9]
(b) [x**2 for x in range(5)]	[0, 1, 4, 9, 16]
(c) nums = [-1, 1, -2, 2, -3, 3, -4, 4] [x for x in nums if x >= 0]	[1, 2, 3, 4]
(d) [ord(ch) for ch in 'Hello']	[72, 101, 108, 108, 111]
(e) vowels = ('a', 'e', 'i', 'o', 'u') w = 'Hello' [ch for ch in w if ch in vowels]	['e', 'o']

FIGURE 4-15 List Comprehensions

In the figure, (a) generates a list of squares of the integers in list [1, 2, 3]. In (b), squares are generated for each value in range(5). In (c), only positive elements of list nums are included in the resulting list. In (d), a list containing the character encoding values in the string 'Hello' is created. Finally, in (e), tuple vowels is used for generating a list containing only the vowels in string w. List comprehensions are a very powerful feature of Python.

LET’S TRY IT
From the Python Shell, enter the following and observe the results.

```
... temperatures 5 [88, 94, 97, 89, 101, 98, 102, 95, 100]
... [t for t in temperatures if t .5 100]
???

... [(t 2 32) * 5/9 for t in temperatures]
???
```

List comprehensions in Python provide a concise means of generating a more varied set of sequences than those that can be generated by the range function.

COMPUTATIONAL PROBLEM SOLVING

4.5 Calendar Year Program

In this section, we extend the calendar month program given in the Computational Problem Solving section of Chapter 3 to display a complete calendar year.

4.5.1 The Problem

The problem is to display a calendar year for any year between 1800 and 2099, inclusive. The format of the displayed year should be as depicted in Figure 4-16.

```
2015

January                     February                    March
             1  2  3         1  2  3  4  5  6  7         1  2  3  4  5  6  7
 4  5  6  7  8  9 10         8  9 10 11 12 13 14         8  9 10 11 12 13 14
11 12 13 14 15 16 17        15 16 17 18 19 20 21        15 16 17 18 19 20 21
18 19 20 21 22 23 24        22 23 24 25 26 27 28        22 23 24 25 26 27 28
25 26 27 28 29 30 31                                    29 30 31

April                       May                         June
          1  2  3  4                     1  2             1  2  3  4  5  6
 5  6  7  8  9 10 11         3  4  5  6  7  8  9         7  8  9 10 11 12 13
12 13 14 15 16 17 18        10 11 12 13 14 15 16        14 15 16 17 18 19 20
19 20 21 22 23 24 25        17 18 19 20 21 22 23        21 22 23 24 25 26 27
26 27 28 29 30              24 25 26 27 28 29 30        28 29 30
                            31

July                        August                      September
          1  2  3  4                        1                1  2  3  4  5
 5  6  7  8  9 10 11         2  3  4  5  6  7  8         6  7  8  9 10 11 12
12 13 14 15 16 17 18         9 10 11 12 13 14 15        13 14 15 16 17 18 19
19 20 21 22 23 24 25        16 17 18 19 20 21 22        20 21 22 23 24 25 26
26 27 28 29 30 31           23 24 25 26 27 28 29        27 28 29 30
                            30 31

October                     November                    December
             1  2  3         1  2  3  4  5  6  7                1  2  3  4  5
 4  5  6  7  8  9 10         8  9 10 11 12 13 14         6  7  8  9 10 11 12
11 12 13 14 15 16 17        15 16 17 18 19 20 21        13 14 15 16 17 18 19
18 19 20 21 22 23 24        22 23 24 25 26 27 28        20 21 22 23 24 25 26
25 26 27 28 29 30 31        29 30                       27 28 29 30 31
```

FIGURE 4-16 Calendar Month Display

4.5.2 Problem Analysis

The computational issues for this problem are similar to the calendar month program of Chapter 3. We need an algorithm for computing the first day of a given month for years 1800–2099. However, since the complete year is being displayed, only the day of the week for January 1st of the given year needs be computed—the rest of the days follow from knowing the number of days in each month (including February for leap years). The algorithm previously developed to display a calendar month, however, is not relevant for this program. Instead, the information will first be stored in a data structure allowing for the months to be displayed three across.

4.5.3 Program Design

Meeting the Program Requirements

We will develop and implement an algorithm that displays the calendar year as

shown in Figure 4-16. We shall request the user to enter the four-digit year to display, with appropriate input error checking.

Data Description

The program needs to represent the year entered, whether it is a leap year, the day of the week for January 1st of the year, and the number of days in each month (accounting for leap years). The names of each of the twelve months will also be stored for display in the calendar year. Given this information, the calendar year can be appropriately constructed and displayed.

We make use of nested lists for representing the calendar year. The data structure will start out as an empty list and will be built incrementally as each new calendar month is computed. The list structures for the calendar year and calendar month are given below,

calendar_year 5 [[*calendar_month*], [*calendar_month*], etc.]]
calendar_month 5 [*week_1, week_2, . . ., week_k*]
Each italicized month is represented as a list of four to six strings, with each string storing a week of the month to be displayed (or a blank line for alignment purposes).
FEBRUARY 2015 MAY 2015
Sun Mon Tues Wed Thur Fri Sat Sun Mon Tues Wed Thur Fri Sat 1 2 3 4 5 6 7 1 2

8 9 10 11 12 13 14 4 3 4 5 6 7 8 9 6 15 16 17 18 19 20 21 lines 10 11 12 13 14 15 16 22 23 24 25 26 27 28 17 18 19 20 21 22 23 lines 24 25 26 27 28 29 30 31

The strings are formatted to contain all the spaces needed for proper alignment when displayed. For example, since the first week of May 2015 begins on a Friday, the string value for this week would be,

' 1 2'
The complete representation for the calendar year 2015 is given below, with the details shown for the months of February and May.
[[*January*],
[' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13 14', February ' 15 16 17 18 19 20 21', ' 22 23 24 25 26 27 28'],
[March],

[April],

[' 1 2', ' 3 4 5 6 7 8 9', May ' 10 11 12 13 14 15 16', ' 17 18 19 20 21 22 23', ' 24 25 26 27 28 29 30', ' 31 '],

[June],
[July],
[August],
[September],
[October],
[November],
[December]]

(Typically, yearly calendars combine the one or two remaining days of the month on the sixth line of a calendar month onto the previous week. We shall not do that in this program, however.) Algorithmic Approach

We make use of the algorithm for determining the day of the week previously used. For this program, however, the only date for which the day of the week needs to be determined is January 1 of a given year. Thus, the original day of the week algorithm can be simplified by removing variable day and replacing its occurrence on line 6 with 1, given in Figure 4-17.

To determine the day of the week for January 1 of a given **year**:

1. Let **century_digits** be equal to the first two digits of the year.
2. Let **year_digits** be equal to the last two digits of the year.
3. Let **value** be equal to **year_digits** + floor(**year_digits** / 4)
4. If **century_digits** equals 18, then add 2 to **value**, else if **century_digits** equals 20, then add 6 to **value**.
5. If **year** is not a leap year then add 1 to **value**.
6. Set **value** equal to (**value** + 1) mod 7.
7. If **value** is equal to 1 (Sunday), 2 (Monday), … 0 (Saturday).

FIGURE 4-17 Simplified Day of the Week Algorithm
Overall Program Steps
The overall steps in this program design are given in Figure 4-18.

4.5.4 Program Implementation and Testing
Stage 1—Determining the Day of the Week (for January 1st)

Wefirst develop and test the code for determining the day of the week for January 1st of a given year. This modified code from the calendar month program is given in Figure 4-19.

FIGURE 4-18 Overall Steps of Calendar Year Program

Line 4 initializes Boolean flag terminate to False. If the user enters −1 for the

year (in **lines 10–13**), terminate is set to True and the while loop at **line 7** terminates, thus terminating the program. If a valid year is entered, **lines 19–42** are executed.

Lines 19–22 determine if the year is a leap year using the same code as in the calendar month program, assigning Boolean variable leap_year accordingly. **Lines 25–40** implement the simplified day of the week algorithm for determining the day of the week for January 1 of a given year in Figure 4.17, with the result displayed on **line 42**.

Stage 1—Testing

We show a sample test run of this stage of the program in Figure 4-20.
Figure 4-21 displays the test cases used for this program.
Since all test cases passed, we can move on to the next stage of program development.

Stage 2—Constructing the Calendar Year Data Structure

Next we develop the part of the program that constructs the data structure holding all of the calendar year information to be displayed. The data structure begins empty and is incrementally built, consisting of nested lists, as previously discussed.

```python
# Calendar Year Program (Stage 1)

# initialization
terminate = False

# prompt for years until quit
while not terminate:

    # get year
    year = int(input('Enter year (yyyy) (-1 to quit): '))

    while (year < 1800 or year > 2099) and year != -1:
        year = int(input('INVALID - Enter year(1800-2099): '))

    if year == -1:
        terminate = True
    else:
        # determine if leap year
        if (year % 4 == 0) and (not (year % 100 == 0) or (year % 400 == 0)):
            leap_year = True
        else:
            leap_year = False

        # determine day of the week
        century_digits = year // 100
        year_digits = year % 100

        value = year_digits + (year_digits // 4)

        if century_digits == 18:
            value = value + 2
        elif century_digits == 20:
            value = value + 6

        # leap year check
        if not leap_year:
            value = value + 1

        # determine first day of month for Jan 1
        first_day_of_month = (value + 1) % 7

        print('Day of week is:', first_day_of_month)
```

FIGURE 4-19 Stage 1 of the Calendar Year Program

```
Enter year (yyyy) (-1 to quit): 1800
Day of week is: 4
Enter year (yyyy) (-1 to quit): 1900
Day of week is: 2
Enter year (yyyy) (-1 to quit): 1984
Day of week is: 1
Enter year (yyyy) (-1 to quit): 1985
Day of week is: 3
Enter year (yyyy) (-1 to quit): 3000
INVALID - Enter year(1800-2099): 2000
Day of week is: 0
Enter year (yyyy) (-1 to quit): 2009
Day of week is: 5
Enter year (yyyy) (-1 to quit): -1
>>>
```

FIGURE 4-20 Example Output of First Stage Testing

Calendar Month	Expected Results first day of month	Actual Results first day of month	Evaluation
January 1800	4 (Wednesday)	4	Passed
January 1900	2 (Monday)	2	Passed
January 1984	1 (Sunday)	1	Passed
January 1985	3 (Tuesday)	3	Passed
January 2000	0 (Saturday)	0	Passed
January 2009	5 (Thursday)	5	Passed

FIGURE 4-21 Test Cases for Stage 1 of the Calendar Year Program

Figure 4-22 shows an implementation of this stage of the program.

Lines 4–14 perform the required initialization. Tuples days_in_month and month_ names have been added to the program to store the number of days for each month (with February handled as an exception) and the month names. On **line 11**, calendar_year is initialized to the empty list. It will be constructed month-by-month for the twelve months of the year. There is the need for strings of blanks of various lengths in the program, initialized as month_ separator, blank_week, and blank_col (**lines 12–14**). The calendar_ year data structure will contain all the space characters needed for the calendar months to be properly displayed. Therefore, there will be no need to develop code that determines how

each month should be displayed as in the calendar month program. The complete structure will simply be displayed row by row.

Lines 17–49 are the same as the first stage of the program for determining the day of the week of a given date. Once the day of the week for January 1st of the given year is known, the days of the week for all remaining dates simply follow. Thus, there is no need to calculate the day of the week for any other date.

Line 52 begins the for loop for constructing each of the twelve months. On **line 53**, the month name is retrieved from tuple month_names and assigned to month_name. Variable current_ day, holding the current day of the month, is initialized to 1 for the new month (**line 56**). In **lines 57–60**, first_day_of_current_month, determined by the day of the week algorithm, is converted to the appropriate column number. Thus, since 0 denotes Saturday, if first_day_of_current_month equals 0, starting_col is set to 7. Otherwise, starting_col is set to first_day_of_current_month (e.g., if first_day_of_ current_month is 1, then starting_col is set to 1).

In **lines 62–64**, the initialization for a new month finishes with the reassignment of current_col, calendar_week, and calendar_month. Each calendar week of a given month is initially assigned to the empty string, with each date appended one-by-one. Variable current_col is used to keep track of the current column (day) of the week, incremented from 0 to 6. Since the first day of the month can fall on any day of the week, the first week of any month may contain blank (“skipped”) columns. This includes the columns from current_col up to but not including starting_col. The while loop in **lines 67–69** appends any of these skipped columns to empty string calendar_week.

Lines **72–75** assign num_days_this_month to the number of days stored in tuple days_ in_month. The exception for February, based on whether the year is a leap year or not, is handled as a special case. The while loop at **line 77** increments variable current_day from 1 to the number of

```python
# Calendar Year Program (Final Version)

# initialization
terminate = False
days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

month_names = ('January', 'February', 'March', 'April', 'May', 'June'
               'July', 'August', 'September', 'October', 'November',
               'December')

calendar_year = []
month_separator = format(' ', '8')
blank_week = format(' ', '21')
blank_col = format(' ', '3')

# prompt for years until quit
while not terminate:

    # get year
    year = int(input('Enter year (yyyy) (-1 to quit): '))
    while (year < 1800 or year > 2099) and year != -1:
        year = int(input('INVALID - Enter year(1800-2099): '))

    if year == -1:
        terminate = True
    else:
        # determine if leap year
        if (year % 4 == 0) and (not (year % 100 == 0) or
            (year % 400 == 0)):
            leap_year = True
        else:
            leap_year = False

        # determine day of the week
        century_digits = year // 100
        year_digits = year % 100
        value = year_digits + (year_digits // 4)

        if century_digits == 18:
            value = value + 2
        elif century_digits == 20:
            value = value + 6

        # leap year check
        if not leap_year:
            value = value + 1

        # determine first day of month for Jan 1
        first_day_of_current_month = (value + 1) % 7

        # construct calendar for all 12 months
        for month_num in range(12):
            month_name = month_names[month_num]

            # init for new month
            current_day = 1
            if first_day_of_current_month == 0:
                starting_col = 7
            else:
                starting_col = first_day_of_current_month

            current_col = 1
            calendar_week = ''
            calendar_month = []

```

FIGURE 4-22 Stage 2 of the Calendar Year Program (*Continued*)

```
66              # add any needed leading space for first week of month
67              while current_col < starting_col:
68                  calendar_week = calendar_week + blank_col
69                  current_col = current_col + 1
70
71              # store month as separate weeks
72              if (month_name == 'February') and leap_year:
73                  num_days_this_month = 29
74              else:
75                  num_days_this_month = days_in_month[month_num]
76
77              while current_day <= num_days_this_month:
78
79                  # store day of month in field of length 3
80                  calendar_week = calendar_week + \
81                     format(str(current_day),'>3')
82
83                  # check if at last column of displayed week
84                  if current_col == 7:
85                      calendar_month = calendar_month + [calendar_week]
86                      calendar_week = ''
87                      current_col = 1
88                  else:
89                      current_col = current_col + 1
90
91                  # increment current day
92                  current_day = current_day + 1
93
94              # fill out final row of month with needed blanks
95              calendar_week = calendar_week + \
96               blank_week[0:(7-current_col+1) * 3]
97              calendar_month = calendar_month + [calendar_week]
98
99              # reset values for next month
100             first_day_of_current_month = current_col
101             calendar_year = calendar_year + [calendar_month]
102             calendar_month = []
103
104     print(calendar_year)
105
106     #reset for another year
107     calendar_year = []
```

FIGURE 4-22 Stage 2 of the Calendar Year Program

days in the month. In **lines 80–81** each date is appended to calendar_week right-justified as a string of length three by use of the format function. Thus, a single-digit date will be appended with two leading blanks, and a double-digit date with one leading blank so that the columns of dates align.

For each new date appended to calendar_week, a check is made on **line 84** as to whether the end of the week has been reached. If the last column of the calendar week has been reached (when column_col equals 7) then the constructed calendar_week string is appended to the calendar_month (**line 85**). In addition,

calendar_week is re-initialized to the empty string, and current_col is reset to 1 (**lines 86–87**). If the last column of the calendar week has not yet been reached, then current_col is simply incremented by 1 (**line 89**). Then, on **line 92**, variable current_day is incremented by 1, whether or not a new week is started.

When the while loop (at **line 77**) eventually terminates, variable current_week holds the last week of the constructed month. Therefore, as with the first week of the month, the last week may contain empty columns. This is handled by **lines 95–97**. Before appending calendar_week to calendar_month, any remaining unfilled columns are appended to it (the reason that these final columns must be blank-filled is because months are displayed side-by-side, and therefore are needed to keep the whole calendar properly aligned),

calendar_week 5 calendar_week 1 blank_week[0:(7-current_col 11) * 3]

Thus, the substring of blank_week produced will end up as an empty string if the value of current_col is 6 (for Saturday, the last column) as it should. **Line 100** sets variable first_day_ of_current_month to current_col since current_col holds the column value of the next column that *would have been* used for the current month, and thus is the first day of the following month. On **line 101**, the completed current month is appended to list calendar_year. And on **line 102**, calendar_month is reset to an empty list in anticipation of the next month to be constructed.

Finally, on **line 104**, the complete calendar_year list is displayed. Because the program prompts the user for other years to be constructed and displayed, the calendar_year list is reset to the empty list (**line 107**).

Stage 2—Testing
The program terminates with an error on line 53,
Enter year (yyyy) (-1 to quit): 2015
Traceback (most recent call last):
File “C:\My Python Programs\CalendarYearStage2.py”, line 54, in ,module .
month_name 5 month_names[month_num]
IndexError: tuple index out of range
This line is within the for loop at line 52,
for month_num in range(12):
month_name 5 month_names[month_num]
For some reason, index variable month_num is out of range for tuple

month_names. We look at the final value of month_num by typing the variable name into the Python shell, ... month_num
11

Since month_names has index values 0–11 (since of length 12), an index value of 11 should not be out of range. How, then, can this index out of range error happen? Just to make sure that month_ names has the right values, we display its length,

.. len(month_names)
11

This is not right! The tuple month_names should contain all twelve months of the year. That is the way it was initialized on line 7, and tuples, unlike lists, cannot be altered, they are immutable. This does not seem to make sense. To continue our investigation, we display the value of the tuple,

.. month_names
('January', 'February', 'March', 'April', 'May', 'JuneJuly', 'August', 'September', 'October', 'November', 'December')
...

Now we see something that doesn’t look right. Months June and July are concatenated into one string value 'JuneJuly’ making the length of the tuple 11, and not 12 (as we discovered). *That* would explain why the index out of range error occurred.

What, then, is the problem. Why were the strings 'June' and 'July' concatenated? We need to look at the line of code that creates this tuple,
month_names 5 ('January', 'February', 'March', 'April', 'May', 'June' 'July', 'August', 'September', 'October', 'November', 'December')

It looks OK. Strings 'June' and 'July' were written as separate strings. We then decide to count the number of items in the tuple. Since items in tuples and lists are separated by commas, we count the number of items between the commas. We count the items up to 'May', which is five items as it should be, then 'June', which is six items . . . ah, there is no comma after the string 'June'! *That* must be why strings 'June' and 'July' were concatenated, and thus the source of the index out of range error. We try to reproduce this in the shell,

... 'June' 'July'
'JuneJuly' That’s it! We have found the problem and should feel good about it. After making the correction and re-executing the program, we get the following results,

```
Enter year (yyyy) (-1 to quit): 2015
[[' 1 2 3', ' 4 5 6 7 8 9 10', ' 11 12 13 14 15 16 17', ' 1 8 19 20,21 22 23 24', ' 25 26
27 28 29 30 31',' '], [' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13 14', ' 15 16 17 18 19 20 21',
' 22 23 24 25 26 27 28', ' '], [' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13 14', ' 15 16 17 18 19
20 21', ' 22 23 24 25 26 27 28', ' 29 30 31

'], [' 1 2 3 4', ' 5 6 7 8 9 10 11', ' 12 13 14 15 16 17 18
', ' 19 20 21 22 23 24 25', ' 26 27 28 29 30 '], [' 1 2', ' 3 4 5 6 7 8 9', ' 10 11 12 13
14 15 16', ' 17 18 19 20 21 22 23', ' 24
25 26 27 28 29 30', ' 31 '], [' 1 2 3 4 5 6', ' 7 8 9 10 11 12 13', ' 14 15 16 17 18 19
20', ' 21 22 23 24 25 26 27', ' 28 29 30 '], [' 1 2 3 4', ' 5 6 7 8 9 10 11', ' 12 13 14
15 16
17 18', ' 19 20 21 22 23 24 25', ' 26 27 28 29 30 31 '], ['
1', ' 2 3 4 5 6 7 8', ' 9 10 11 12 13 14 15', ' 16 17 18 19 20 21 22', ' 23 24 25 26 27
28 29', ' 30 31 '], [' 1 2 3 4 5', ' 6 7 8 9 10 11 12', ' 13 14 15 16 17 18 19', ' 20 21
22 23 24 25 26', ' 27 28
29 30 '], [' 1 2 3', ' 4 5 6 7 8 9 10', ' 11 12 13
14 15 16 17', ' 18 19 20 21 22 23 24', ' 25 26 27 28 29 30 31', ' '], [' 1 2 3 4 5 6 7',
' 8 9 10 11 12 13 14', ' 15 16 17 18 19 2
0 21', ' 22 23 24 25 26 27 28', ' 29 30 '], [' 1 2 3 4 5', ' 6 7 8 9 10 11 12', ' 13 14 15
16 17 18 19', ' 20 21 22 23 24 25 26', ' 27 28 29 30 31 ']]
Enter year (yyyy) (-1 to quit):
```

We can see if the output looks like the structure that we expect. The first item in the list, the structure for the month of January, is as follows,

```
[[' 1 2 3', ' 4 5 6 7 8 9 10', ' 11 12 13 14 15 16 17', ' 18 19 20 21 22 23 24', ' 25 26
27 28 29 30 31', ' ']
```

In checking against available calendar month calculators, we see that the first day of the month for January 2015 is a Thursday. Thus, the first week of the month should have four skipped days, followed by 1, 2, and 3 each in a column width of 3. We find that there are fourteen blank characters in the first line. The first twelve are for the four skipped columns, and the last two are for the right-justified string ‘1’ in the column of the first day of the month,

' 1 2 3'
\|||/
12 blank 2 blank
chars chars

Since there are five weeks in the month, there should be one extra “blank week” at the end of the list to match the vertical spacing of all other months. We see, in fact, that the last (sixth) string is a string of blanks.

Since the calendar_year structure looks correct, we now develop the final stage of the program that displays the complete calendar year.
Stage 3—Displaying the Calendar Year Data Structure

We now give the complete calendar year program in Figure 4-23. In this final version, the only change at the start of the program is that a program greeting is added on **line 19**. The rest of the program is the same up to **line 105**, the point where the calendar year has been constructed. (The print(calendar_year) line and re-initialization of calendar_year to the empty list have been removed from the previous version, since they were only there for testing purposes.)

The new code in this version of the program is in **lines 107–141**, which displays the calendar year. The calendar year output is given in Figure 4-24.
On **line 108** the year is displayed. Because the months are displayed three across, as shown in Figure 4-16, the for loop on **line 111** iterates variable month_num over the values [0, 3, 6, 9]. Thus, when month_num is 0, months 0-2 (January–March) are displayed. When month_num is 3, months 3-5 (April–June) are displayed, and so forth.
The for loop at **line 114** displays the month names for each row (for example, January, February, and March). Each is displayed left-justified in a field width of 19. A leading blank character is appended to the formatting string to align with the first column of numbers displayed for each month. The print(. . ., end 5") form of print is used, which prevents the cursor from moving to the next line. Thus, the months can be displayed side-by-side. Variable month_ separator contains the appropriate number of blank spaces (initialized at the top of the program) to provide the required amount of padding between the months, as shown below,

January February March 1 2 3 1 2 3 4 5 6 7 1 2 3 4 5 6 7
\|||||/ \|||||||/
month_separator month_separator

Lines 119–120 perform the initialization needed for the following while loop (at **line 122**), which displays each week, one-by-one, of the current three months. Variable week is initialized to zero for each month and is used to keep count of the number of weeks displayed. Variable lines_to_print is initialized to True to start the execution of the following while loop.

At **line 125** within the while loop, lines_to_print is initialized to False. It is then set to True by any (or all) of the current three months being displayed only if they still have more calendar lines (weeks) to print, thus causing the while loop to continue with another

```python
# Calendar Year Program (Final Version)

# initialization
terminate = False
days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

month_names = ('January', 'February', 'March', 'April', 'May', 'June',
               'July', 'August', 'September', 'October', 'November',
               'December')

calendar_year = []
month_separator = format(' ', '8')
blank_week = format(' ', '21')
blank_col = format(' ', '3')

# prompt for years until quit
while not terminate:

    # program greeting
    print ('This program will display a calendar year for a given year')

    # get year
    year = int(input('Enter year (yyyy) (-1 to quit): '))
    while (year < 1800 or year > 2099) and year != -1:
        year = int(input('INVALID - Enter year(1800-2099): '))

    if year == -1:
        terminate = True
    else:
        # determine if leap year
        if (year % 4 == 0) and (not (year % 100 == 0) or
            (year % 400 == 0)):
            leap_year = True
        else:
            leap_year = False

        # determine day of the week
        century_digits = year // 100
        year_digits = year % 100
        value = year_digits + (year_digits // 4)

        if century_digits == 18:
            value = value + 2
        elif century_digits == 20:
            value = value + 6

        # leap year check
        if not leap_year:
            value = value + 1

        # determine first day of month for Jan 1
        first_day_of_current_month = (value + 1) % 7

        # construct calendar for all 12 months
        for month_num in range(12):
            month_name = month_names[month_num]

            # init for new month
            current_day = 1
            if first_day_of_current_month == 0:
                starting_col = 7
            else:
                starting_col = first_day_of_current_month

```

```
            current_col = 1
            calendar_week = ''
            calendar_month = []

            # add any needed leading space for first week of month
            while current_col < starting_col:
                calendar_week = calendar_week + blank_col
                current_col = current_col + 1

            # store month as separate weeks
            if (month_name == 'February') and leap_year:
                num_days_this_month = 29
            else:
                num_days_this_month = days_in_month[month_num]

            while current_day <= num_days_this_month:

                # store day of month in field of length 3
                calendar_week = calendar_week + \
                   format(str(current_day),'>3')

                # check if at last column of displayed week
                if current_col == 7:
                    calendar_month = calendar_month + [calendar_week]
                    calendar_week = ''
                    current_col = 1
                else:
                    current_col = current_col + 1

                # increment current day
                current_day = current_day + 1

            # fill out final row of month with needed blanks
            calendar_week = calendar_week + \
                            blank_week[0:(7-current_col+1) * 3]
            calendar_month = calendar_month + [calendar_week]

            # reset values for next month
            first_day_of_current_month = current_col
            calendar_year = calendar_year + [calendar_month]
            calendar_month = []

        # print calendar year
        print('\n', year,'\n')

        # each row starts with January, April, July, or October
        for month_num in [0,3,6,9]:

            # displays three months in each row
            for i in range(month_num, month_num + 3):
                  print(' ' + format(month_names[i],'19'),
                        month_separator, end='')

            # display each week of months on separate lines
            week = 0
            lines_to_print = True

```

```
122             while lines_to_print:
123
124                 # init
125                 lines_to_print = False
126
127                 # another week to display for first month in row?
128                 for k in range(month_num, month_num + 3):
129                     if week < len(calendar_year[k]):
130                         print(calendar_year[k][week], end='')
131                         lines_to_print = True
132                     else:
133                         print(blank_week, end='')
134
135                     print(month_separator, end='')
136
137                 # move to next screen line
138                 print()
139
140                 # increment week
141                 week = week + 1
```

FIGURE 4-23 Final Stage of the Calendar Year Program

iteration. This occurs within the for loop at **lines 128–135**. Since variable month_num indicates the current month being displayed, the number of weeks in the month is determined by the length of the tuple of strings for the current month k.

len(calendar_year[k])
Note that some months may have no more weeks to display, whereas others may. This is the case for the first three months of 2015,
January February March

1 2 3 1 2 3 4 5 6 7 1 2 3 4 5 6 7
4 5 6 7 8 9 10 8 9 10 11 12 13 14 8 9 10 11 12 13 14 11 12 13 14 15 16 17 15 16 17 18 19 20 21 15 16 17 18 19 20 21
18 19 20 21 22 23 24 22 23 24 25 26 27 28 22 23 24 25 26 27 28
25 26 27 28 29 30 31 29 30 31

In this case, the while loop needs to continue to iterate in order to display the last lines of January and March even though the last line of February has been displayed. Therefore, in cases where a given month has a line to print but another month doesn’t, a blank line is displayed in order to maintain the correct alignment of month weeks. After the week of dates (or blank week) is output for each of the three months, the cursor is moved to the start of the next line (on **line 138**) and variable week is incremented by one (**line 141**) before the loop begins

the next iteration for displaying the next row of calendar weeks.

Chapter Summary 161

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
>>>
This program will display a calendar year for a given year
Enter year (yyyy) (-1 to quit): 2015

 2015

 January                        February                       March
                 1  2  3         1  2  3  4  5  6  7          1  2  3  4  5  6  7
  4  5  6  7  8  9 10          8  9 10 11 12 13 14          8  9 10 11 12 13 14
 11 12 13 14 15 16 17         15 16 17 18 19 20 21         15 16 17 18 19 20 21
 18 19 20 21 22 23 24         22 23 24 25 26 27 28         22 23 24 25 26 27 28
 25 26 27 28 29 30 31                                      29 30 31


 April                          May                            June
              1  2  3  4                       1  2              1  2  3  4  5  6
  5  6  7  8  9 10 11          3  4  5  6  7  8  9          7  8  9 10 11 12 13
 12 13 14 15 16 17 18         10 11 12 13 14 15 16         14 15 16 17 18 19 20
 19 20 21 22 23 24 25         17 18 19 20 21 22 23         21 22 23 24 25 26 27
 26 27 28 29 30               24 25 26 27 28 29 30         28 29 30
                              31

 July                           August                         September
              1  2  3  4                          1                 1  2  3  4  5
  5  6  7  8  9 10 11          2  3  4  5  6  7  8          6  7  8  9 10 11 12
 12 13 14 15 16 17 18          9 10 11 12 13 14 15         13 14 15 16 17 18 19
 19 20 21 22 23 24 25         16 17 18 19 20 21 22         20 21 22 23 24 25 26
 26 27 28 29 30 31            23 24 25 26 27 28 29         27 28 29 30
                              30 31

 October                        November                       December
                 1  2  3         1  2  3  4  5  6  7                 1  2  3  4  5
  4  5  6  7  8  9 10          8  9 10 11 12 13 14          6  7  8  9 10 11 12
 11 12 13 14 15 16 17         15 16 17 18 19 20 21         13 14 15 16 17 18 19
 18 19 20 21 22 23 24         22 23 24 25 26 27 28         20 21 22 23 24 25 26
 25 26 27 28 29 30 31         29 30                        27 28 29 30 31


Enter year (yyyy) (-1 to quit): -1
>>> |
                                                                    Ln: 152 Col: 4
```

FIGURE 4-24 Calendar Year Program Output

Finally, the while loop at line 122 continues to iterate until there are no more lines to display for all of the three months currently being displayed—that is, until lines_to_print is False. Figure 4-25 displays the results of testing this final version by using the test plan from the Calendar Month program of Chapter 3. The test plan passed for all test cases.

CHAPTER SUMMARY General Topics

Linear Data Structures List Operations
List Traversal
The Empty List and Its Use Nested Lists
List Iteration
Loop Variable/Index Variable

Calendar Month	Expected Results		Actual Results		Evaluation
	first day	num days	first day	num days	
January 1800	Wednesday	31	Wednesday	31	Pass
February 1800	Saturday	28	Saturday	28	Pass
April 1860	Sunday	30	Sunday	30	Pass
July 1887	Friday	31	Friday	31	Pass
February 1904	Monday	29	Monday	29	Pass
March 1945	Thursday	31	Thursday	31	Pass
September 1960	Thursday	30	Thursday	30	Pass
October 1990	Monday	31	Monday	31	Pass
November 1992	Sunday	30	Sunday	30	Pass
February 2000	Tuesday	29	Tuesday	29	Pass
August 2006	Tuesday	31	Tuesday	31	Pass
May 2014	Thursday	31	Thursday	31	Pass
December 2019	Sunday	31	Sunday	31	Pass
June 2084	Thursday	30	Thursday	30	Pass

FIGURE 4-25 Final Calendar Year Program Testing

Python-Specific Programming Topics

Lists in Python
List Operations in Python
Empty Lists and Tuples in Python
Lists, Tuples, and Strings as Sequences in Python Additional Sequence Operations

Nested Lists and Tuples in Python
For/ While Loops and List Iteration in Python Built-in Range Function in Python
Iterating over List (Sequence) Elements vs.

Iterating over Index Values in Python Assigning Lists in Python
List Comprehensions in Python

CHAPTER EXERCISES Section 4.1
1. (a) Give the index values of all the odd numbers in the following list representation, assuming zero-based indexing.

23
16
14
33
19
6
11

Chapter Exercises 163
(b) How many elements would be looked at when the list is traversed (from top to bottom) until the value 19 was found?
Section 4.2
2. Which of the following lists are syntactically correct in Python?
(a) [1, 2, 3, 'four'] **(b)**[1, 2, [3, 4]] **(c)**[[1, 2, 3]['four']] **3.** For lst 5[4, 2, 9, 1], what is the result of each of the following list operations? **(a)** lst[1] **(b)** lst.insert(2, 3) **(c)** del lst[3] **(d)** lst.append(3)

4. For fruit 5['apple', 'banana', 'pear', 'cherry'], use a list operation to change the list to ['apple', 'banana', 'cherry'] .
5. For a list of integers, lst, give the code to retrieve the maximum value of the second half of the list.
6. For variable product_code containing a string of letters and digits,
(a) Give an if statement that outputs “Verified” if product_code contains both a “Z” and a “9”, and outputs “Failed” otherwise.
(b) Give a Python instruction that prints out just the last three characters in product_code.
7. Which of the following are valid operations on tuples (for tuples t1 and t2)?
(a)len(t1) **(b)**t1 1 t2 **(c)**t1.append(10) **(d)**t1.insert(0, 10)

8. For str1 5'Hello World', answer the following,
(a) Give an instruction that prints the fourth character of the string.
(b) Give an instruction that finds the index location of the first occurrence of the letter 'o' in the string.
9. For a nested list lst that contains sublists of integers of the form [n1, n2, n3],
(a) Give a Python instruction that determines the length of the list.
(b) Give Python code that determines how many total integer values there are in list lst. **(c)** Give Python code that totals all the values in list lst.
(d) Given an assignment statement that assigns the third integer of the fourth element (sublist) of lst to the value 12.

Section 4.3

10. For a list of integers named nums,
(a) Write a while loop that adds up all the values in nums.
(b) Write a for loop that adds up all the values in nums in which the loop variable is assigned each value

in the list.
(c) Write a for loop that adds up all the elements in nums in which the loop variable is assigned to the
index value of each element in the list.
(d) Write a for loop that displays the elements in nums backwards.
(e) Write a for loop that displays every other element in nums, starting with the first element.

Section 4.4
11. For list1 5 [1, 2, 3, 4] and list2 5[5, 6, 7, 8], give the values of list1[0] and list2[0] where indicated after the following assignments.
(a) list1[0] 510

list2[0] 550 list1[0] _____ list2[0] _____ **(b)** list2 5list1 list1[0] _____ list2[0] _____ **(c)** list2[0] 515 list1[0] _____ list2[0] _____ **(d)** list1[0] 50 list1[0] _____ list2[0] _____

12. Give an appropriate list comprehension for each of the following.
(a) Producing a list of consonants that appear in string variable w.
(b) Producing a list of numbers between 1 and 100 that are divisible by 3.
(c) Producing a list of numbers, zero_values, from a list of floating-point values,

data_values,

that are within some distance, epsilon, from 0.

PYTHON PROGRAMMING EXERCISES

P1. Write a Python program that prompts the user for a list of integers, stores in another list only those values between 1–100, and displays the resulting list.

P2. Write a Python program that prompts the user for a list of integers, stores in another list only those values that are in tuple valid_values, and displays the resulting list.

P3. Write a Python program that prompts the user for a list of integers and stores them in a list. For all values that are greater than 100, the string 'over' should be stored instead. The program should display the resulting list.

P4. Write a Python program that prompts the user to enter a list of first names and stores them in a list. The program should display how many times the letter 'a' appears within the list.

P5. Write a Python program that prompts the user to enter a list of words and stores in a list only those words whose first letter occurs again within the word (for example, 'Baboon'). The program should display the resulting list.

P6. Write a Python program that prompts the user to enter types of fruit, and how many pounds of fruit there are for each type. The program should then display the information in the form *fruit, weight* listed in alphabetical order, one fruit type per line as shown below,

Apple, 6 lbs. Banana, 11 lbs. etc.

P7. Write a Python program that prompts the user to enter integer values for each of two lists. It then should displays whether the lists are of the same length, whether the elements in each list sum to the same value, and whether there are any values that occur in both lists.

PROGRAM MODIFICATION PROBLEMS

M1. Chinese Zodiac Program: Japanese and Vietnamese Variations

Modify the Chinese Zodiac program in the chapter to allow the user to select the Chinese Zodiac, the Japanese Zodiac, or the Vietnamese Zodiac. The Japanese Zodiac is the same as the Chinese Zodiac, except that “Pig” is substituted with “Wild Boar.” The Vietnamese Zodiac is also the same except that the “Ox” is substituted with “Water Buffalo” and “Rabbit” is replaced with “Cat.”

M2. Chinese Zodiac Program: Improved Accuracy
The true Chinese Zodiac does not strictly follow the year that a given person was born. It also depends on the month and date as well, which vary over the years. Following are the correct range of dates for each of the Zodiac symbols for the years 1984 to 2007 (which includes two full cycles of the zodiac). Modify

鼠 Rat	Feb 02, **1984** – Feb 19, 1985
牛 Ox	Feb 20, **1985** – Feb 08, 1986
虎 Tiger	Feb 09, **1986** – Jan 28, 1987
兔 Rabbit	Jan 29, **1987** – Feb 16, 1988
龍 Dragon	Feb 17, **1988** – Feb 05, 1989
蛇 Snake	Feb 06, **1989** – Jan 26, 1990
馬 Horse	Jan 27, **1990** – Feb 14, 1991
羊 Sheep	Feb 15, **1991** – Feb 03, 1992
猴 Monkey	Feb 04, **1992** – Jan 22, 1993
鷄 Rooster	Jan 23, **1993** – Feb 09, 1994
狗 Dog	Feb 10, **1994** – Jan 30 1995
猪 Pig	Jan 31, **1995** – Feb 18, 1996

鼠 Rat	Feb 19, **1996** – Feb 06, 1997
牛 Ox	Feb 07, **1997** – Jan 27, 1998
虎 Tiger	Jan 28, **1998** – Feb 15, 1999
兔 Rabbit	Feb 16, **1999** – Feb 04, 2000
龍 Dragon	Feb 05, **2000** – Jan 23, 2001
蛇 Snake	Jan 24, **2001** – Feb 11, 2002
馬 Horse	Feb 12, **2002** – Jan 31, 2003
羊 Sheep	Feb 01, **2003** – Jan 21, 2004
猴 Monkey	Jan 22, **2004** – Feb 08, 2005
鷄 Rooster	Feb 09, **2005** – Jan 28, 2006
狗 Dog	Jan 29, **2006** – Feb 17, 2007
猪 Pig	Feb 18, **2007** – Feb 06, 2008

the Chinese Zodiac program in the chapter so that the user is prompted to enter their date of birth, including month and day, and displays the name and characteristics of the corresponding Chinese Zodiac symbol based on the more accurate zodiac provided here.

M3. Password Encryption/Decryption Program: Multiple Executions
Modify the Password Encryption/Decryption program in the chapter so that it allows the user to continue to encrypt and decrypt passwords until they quit.

M4. Password Encryption/Decryption Program: Secure Password Check
Modify the Password Encryption/Decryption program in the chapter so that the program rejects any entered password for encryption that is not considered "secure" enough. A password is considered secure if it contains at least eight characters, with at least one digit and one special character (!, #, etc).

M5. Password Encryption/Decryption Program: Random Key Generation
Modify the Encryption/Decryption program in the chapter so that a new

encryption key is randomly generated each time the program is executed. (See the Python 3 Programmers' Reference for information on the Random module.)

M6. Calendar Year Program: Multilingual Version
Modify the Calendar Year program so that the user can select the language with which the calendar months are labeled. Give the user the choice of at least three different languages from which to select. Find the month names for the other languages online.

M7. Calendar Year Program: Flexible Calendar Format
The program as is always displays three months per row. Modify the Calendar Year program so that the user can select how many months are displayed per row. Allow the user to select either two, three, or four months per row.

PROGRAM DEVELOPMENT PROBLEMS

D1. Morse Code Encryption/Decryption Program
Develop and test a Python program that allows a user to type in a message and have it converted into Morse code, and also enter Morse code and have it converted back to the original message. The encoding of Morse code is given below.

A	• –	N	– •
B	– • • •	O	– – –
C	– • – •	P	• – – •
D	– • •	Q	– – • –
E	•	R	• – •
F	• • – •	S	• • •
G	– – •	T	–
H	• • • •	U	• • –
I	• •	V	• • • –
J	• – – –	W	• – –
K	– • –	X	– • • –
L	• – • •	Y	– • – –
M	– –	Z	– – • •

Format the original message (containing English words) so that there is one sentence per line. Format the Morse code file (containing dots and dashes) so that there is one letter per line, with a blank line following the last letter of each word, and two blank lines following the end of each sentence (except the last).

D2. Holidays Calendar
Develop and test a Python program that displays the day of the week that the following holidays fall on for a year entered by the user,

♦ New Year’s Eve

♦ Valentine’s Day
♦ St. Patrick’s Day
♦ April Fool’s Day
♦ Fourth of July
♦ Labor Day
♦ Halloween
♦ User’s Birthday

Note that Labor Day, as opposed to the other holidays above, does not fall on the same date each year. It occurs each year on the first Monday of September.

D3. The Game of Battleship
Battleship is a game involving ships at sea for each of two players. The ships are located in a grid in which each column of the grid is identified by a letter, and each row by a number, as shown below.

The top half of the board contains the ships of player 1, and the bottom half the ships of player 2. The darkened areas indicate the size and location of ships. Each player starts with the same number and types of ships. The location of each

player's ships is determined by the player. Players take turns taking a shot at the opponent's ships by "calling out" a particular grid location. For example, if player 1 calls out "C10," no ship would be hit in this example. If, however, they were to call out "G10," then player 2's ship (on the bottom half of the board) would be hit. Each player calls out "hit" or "miss" when they are shot at by the other player. When all grid locations of a given ship have been hit, the ship is sunk, and the opponent gets the number of points based on the ship's size (given below).

The number of grid locations that a given ship takes up indicates its type and point value. A typical set of ships is given below.

Type of Ship	Size of Ships
aircraft carrier	5
battleship	4
cruiser	3
submarine	3
destroyer	2

Develop and test a Python program that can play the game of battleship. The user should be able to select the skill level. The higher the skill level, the larger the grid that is created for play. All games start with exactly one of each type of ship for each player. The locations of the computer's ships will be randomly placed. The user, however, must be able to enter the location of each of their ships. The computer's shots into the opponent's grid area should be randomly generated.

D4. Heuristic Play for the Game of Battleship
A heuristic is a general "rule of thumb" for solving a problem. Modify the Game of Battleship program from the previous problem so that the locations of the shots that the computer makes into the opponent's grid area are based on heuristics, rather than being randomly generated. Include an explanation of the heuristics developed.

CHAPTER 5 Functions

Up until this point, we have viewed a computer program as a single series of instructions. Most programs, however, consist of distinct groups of instructions,

each of which accomplishes a specific task. Such a group of instructions is referred to as a "routine." Program routines, called "functions" in Python, are fundamental building blocks in software development. We take our first look at functions in this chapter.

OBJECTIVES

After reading this chapter and completing the exercises, you will be able to: ♦ Explain the concept of a program routine
♦ Explain the concept of parameter passing
♦ Explain the concept of value-returning and non-value-returning functions ♦ Explain the notion of the side-effects of a function call
♦ Differentiate between local scope and global scope
♦ Define and use functions in Python
♦ Explain the concept of keyword and default arguments in Python ♦ Write a Python program using programmer-defined functions ♦ Effectively use trace statements for program testing

CHAPTER CONTENTS

Motivation
Fundamental Concepts
5.1 Program Routines
5.2 More on Functions
Computational Problem Solving 5.3 Credit Card Calculation Program

MOTIVATION

So far, we have limited ourselves to using only the most fundamental features of Python—variables, expressions, control structures, input/print, and lists. In theory, these are the only instructions needed to write any program (that is, to perform any computation). From a *practical* point-of-view, however, these instructions alone are not enough.

The problem is one of complexity. Some smart phones, for example, contain over 10 million lines of code (see Figure 5-1). Imagine

the effort needed to develop and debug software of that size. It certainly cannot be implemented by

any one person, it takes a team of programmers to develop such a project.

In order to manage the complexity of a large problem, it is broken down into smaller subproblems. Then, each subproblem can be focused on and solved separately. In programming, we do the same thing. Programs are divided into manageable pieces called *program routines* (or simply *routines*). Doing so is a form of *abstraction* in which a more general, less detailed view of a system can be achieved. In addition, program routines provide the opportunity for code reuse, so that systems do not have to be created from “scratch.” Routines, therefore, are a fundamental building block in software development.

In this chapter, we look at the definition and use of program routines in Python.

Term	Number of Lines of Code (LOC)	Equivalent Storage
KLOC	1,000	Application programs
MLOC	1,000,000	Operating systems / smart phones
GLOC	1,000,000,000	Number of lines of code in existence for various programming languages

FIGURE 5-1 Measures of Lines of Program Code

FUNDAMENTAL CONCEPTS

5.1 Program Routines

We first introduce the notion of a program routine. We then look in particular at program routines in Python, called *functions*. We have already been using Python's built-in functions such as len, range, and others. We now look more closely at how functions are used in Python, as well as how to define our own.

5.1.1 What Is a Function Routine?

A **routine** is a named group of instructions performing some task. A routine can be **invoked**(*called*) as many times as needed in a given program, as shown in Figure 5-2.

FIGURE 5-2

Program Routine

When a routine terminates, execution automatically returns to the point from which it was called. Such routines may be predefined in the programming language, or designed and implemented by the programmer.

A **function** is Python's version of a program routine. Some functions are designed to return a value, while others are designed for other purposes. We look at these two types of functions next.

A program **routine** is a named group of instructions that accomplishes some task. A routine may be **invoked** (called) as many times as needed in a given program. A **function** is Python's version of a program routine.

5.1.2 Defining Functions

In addition to the built-in functions of Python, there is the capability to define

new functions. Such functions may be generally useful, or specific to a particular program. The elements of a function definition are given in Figure 5-3.

FIGURE 5-3 Example of Python Function Definition

The first line of a function definition is the *function header*. A function header starts with the keyword def, followed by an identifier (avg), which is the function's name. The function name is followed by a comma-separated (possibly empty) list of identifiers (n1, n2, n3) called **formal parameters**, or simply "parameters." Following the *parameter list* is a colon ( ). Following the function header is the body of the function, a suite (program block) containing the function's instructions. As with all suites, the statements must be indented at the same level, relative to the function header.

The number of items in a parameter list indicates the number of values that must be passed to the function, called **actual arguments** (or simply "arguments"), such as the variables num1, num2, and num3 below.

... num1 5 10
... num2 5 25
... num3 5 16

... avg(num1,num2,num3)
Functions are generally defined at the top of a program. However, *every function must be defined before it is called*.
We discuss more about function definition and use in the following sections.

Actual arguments , or simply "arguments," are the values passed to functions to be operated on. **Formal parameters**, or simply "parameters," are the "placeholder" names for the arguments passed.

Value-Returning Functions

A **value-returning function** is a program routine called for its return value, and is therefore similar to a mathematical function. Take the simple mathematical

function f(x) 52x. In this notation, “x” stands for any numeric value that function f may be applied to, for example, f(2)5 2x 54. Program functions are similarly used, as illustrated in Figure 5-4.

Function avg takes three arguments (n1, n2, and n3) and returns the average of the three. The *function call*avg(10, 25, 16), therefore, is an expression that evaluates to the returned function value. This is indicated in the function’s *return statement* of the form return *expr*, where *expr* may be any expression. Next, we look at a second form of program routine called for a purpose other than a returned function value.

FIGURE 5-4 Call to Value-Returning Function
LET’S TRY IT

From the Python Shell, first enter the following function, making sure to indent the code as given. Hit return twice after the last line of the function is entered. Then enter the following function calls and observe the results.

... def avg(n1, n2, n3): ... avg(40, 10, 25) return (n1 1 n2 1 n3) / 3.0 ???
... avg(10, 25, 40) ... avg(40, 25, 10) ??? ???
A **value-returning function** in Python is a program routine called for its return value, and is therefore similar to a mathematical function.
Non-Value-Returning Functions

A **non-value-returning function** is called not for a returned value, but for its *side effects*. A **side effect** is an action other than returning a function value, such as displaying output on the screen. There is a fundamental difference in the way that value-returning and non-value-returning functions are called. A call to a

value-returning function is an expression, as for the call to function avg: result 5 **avg(10, 25, 16)** * factor.

When non-value-returning functions are called, however, the function call is a *statement*, as shown in Figure 5-5. Since such functions do not have a return value, it is incorrect to use a call to a non-value-returning function as an expression.

FIGURE 5-5 Call to Non-Value-Returning Function

In this example, function displayWelcome is called only for the side-effect of the screen output produced. Finally, every function in Python is technically a value-returning function since *any function that does not explicitly return a function value (via a return statement) automatically returns the special value*None. We will, however, consider such functions as non-value-returning functions.

LET’S TRY IT
From the Python Shell, first enter the following function, making sure to indent the code as given. Then enter the following function calls and observe the results.
... def hello(name): ... name 5 'John' print('Hello', name 1 '!') ... hello(name)
???
A **non-value-returning function** is a function called for its *side effects*, and not for a returned function value.
5.1.3 Let’s Apply It—Temperature Conversion Program
(Function Version)

The following is a program (Figure 5-7) that allows a user to convert a range of values from Fahrenheit to Celsius, or Celsius to Fahrenheit, as presented in Chapter 3. In this version, however, the program is designed with the use of functions. This program utilizes the following programming features.

➤ value-returning functions ➤ non-value-returning functions

Example execution of the program is given in Figure 5-6.

```
This program will convert a range of temperatures
Enter (F) to convert Fahrenheit to Celsius
Enter (C) to convert Celsius to Fahrenheit

Enter selection: F
Enter starting temperature to convert: 65
Enter ending temperature to convert: 95

  Degrees  Degrees
Fahrenheit Celsius
   65.0     18.3
   66.0     18.9
   67.0     19.4
   68.0     20.0
   69.0     20.6
   70.0     21.1
   71.0     21.7
   72.0     22.2
   73.0     22.8
   74.0     23.3
   75.0     23.9
   76.0     24.4
   77.0     25.0
   78.0     25.6
   79.0     26.1
   80.0     26.7
   81.0     27.2
   82.0     27.8
   83.0     28.3
   84.0     28.9
   85.0     29.4
   86.0     30.0
   87.0     30.6
   88.0     31.1
   89.0     31.7
   90.0     32.2
   91.0     32.8
   92.0     33.3
   93.0     33.9
   94.0     34.4
   95.0     35.0
```

FIGURE 5-6 Execution of Temperature Conversion Program

```
1  # Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)
2
3  def displayWelcome():
4
5     print('This program will convert a range of temperatures')
6     print('Enter (F) to convert Fahrenheit to Celsius')
7     print('Enter (C) to convert Celsius to Fahrenheit\n')
8
9  def getConvertTo():
10
11    which = input('Enter selection: ')
12    while which != 'F' and which != 'C':
13       which = input('Enter selection: ')
14
15    return which
16
17 def displayFahrenToCelsius(start, end):
18
19     print('\n  Degrees', ' Degrees')
20     print('Fahrenheit', 'Celsius')
21
22     for temp in range(start, end + 1):
23        converted_temp = (temp - 32) * 5/9
24        print('  ', format(temp, '4.1f'), '   ', format(converted_temp, '4.1f'))
25
26 def displayCelsiusToFahren(start, end):
27
28     print('\n  Degrees', ' Degrees')
29     print('  Celsius', 'Fahrenheit')
30
31     for temp in range(start, end + 1):
32        converted_temp = (9/5 * temp) + 32
33        print('  ', format(temp, '4.1f'), '   ', format(converted_temp, '4.1f'))
34
35 # ---- main
36
37 # Display program welcome
38 displayWelcome()
39
40 # Get which converion from user
41 which = getConvertTo()
42
43 # Get range of temperatures to convert
44 temp_start = int(input('Enter starting temperature to convert: '))
45 temp_end = int(input('Enter ending temperature to convert: '))
46
47 # Display range of converted temperatures
48 if which == 'F':
49    displayFahrenToCelsius(temp_start, temp_end)
50 else:
51    displayCelsiusToFahren(temp_start, temp_end)
```

FIGURE 5-7 Temperature Conversion Program (Function Version)

In **lines 3–29** are defined functions displayWelcome, getConvertTo, displayFahrenToCelsius, and displayCelsiusToFahren. The functions are directly called from the main module of the program in **lines 32–48**.

On **line 35**, the non-value-returning function displayWelcome is called. Its job is to display information about the program to the user. It does not need to be passed any arguments since it performs the same output each time it is called. Next, on **line 38**, value-returning function getConvertTo is called. This function also is not passed any arguments. It simply asks the user to enter either 'F' or 'C' to indicate whether they want to convert from Fahrenheit to Celsius, or Celsius to Fahrenheit. The input value entered is returned as the function value.

The instructions on **line 41–42** then prompt the user for the start and end range of temperatures to be converted. This task does not warrant the construction of a function since there are only two input instructions to accomplish this.

The final part of the program displays the converted range of temperatures. Two non-valuereturning functions are defined for accomplishing this task —displayFahrenToCelsius and displayCelsiusToFahren. Each is passed two arguments, temp_start and temp_end, which indicate the range of temperature values to be converted.

What is left to look at is the implementation of each of the individual functions. The implementation of function displayWelcome (**lines 3–6**) is very straightforward. It simply contains three print instructions. Function getConvertTo (**lines 8–13**) contains a call to input followed by a while loop that performs input validation. The user is forced to enter either 'F' or 'C', and is continually prompted to re-enter as long as a value other than these two values is entered. When the loop terminates, variable which is returned by the return statement in **line 13**.

Function displayFahrenToCelsius (**lines 15–21**) and function displayCelsiusToFahren (**lines 23–29**) are similar in design. Each contains two parameters—start and end (which are each passed actual arguments temp_start and temp_end in the main section of the program). Each first prints the appropriate column headings followed by a for statement that iterates variable temp over the requested temperature range. The conversion formula is different in each, however. Each has the same final print instruction to print out the original temperature and the converted temperature in each of the columns.

Self-Test Questions

1. The values passed in a given function call in Python are called,

(a) formal parameters
(b) actual arguments

2. The identifiers of a given Python function providing names for the values passed to it are called, **(a)** formal parameters
(b) actual arguments

3. Functions can be called as many times as needed in a given program. (TRUE/FALSE) **4.** When a given function is called, it is said to be,
(a) subrogated **(b)** invoked **(c)** activated
5. Which of the following types of functions must contain a return statement, **(a)** value-returning functions **(b)** non-value-returning functions
6. Value-returning function calls are,
(a) expressions **(b)** statements
7. Non-value-returning function calls are,
(a) expressions **(b)** statements
8. Which of the following types of routines is meant to produce side effects? **(a)** value-returning functions **(b)** non-value-returning functions
ANSWERS: 1. (b), 2. (a), 3. TRUE, 4. (b), 5. (a), 6. (a), 7. (b), 8. (b)

5.2 More on Functions

In this section we further discuss issues related to function use, including more on function invocation and parameter passing.

5.2.1 Calling Value-Returning Functions

Calls to value-returning functions can be used anywhere that a function's return value is appropriate,
result 5 max(num_list) * 100
Here, we apply built-in function max to a list of integers, num_list. Examples of additional allowable forms of function calls are given below.

(a) result 5 max(num_list1) * max(num_list2)
(b) result 5 abs(max(num_list))
(c) if max(num_list) , 10:...
(d) print('Largest value in num_list is ', max(num_list))

The examples demonstrate that an expression may contain multiple function calls, as in (a); a function call may contain function calls as arguments, as in (b); conditional expressions may contain function calls, as in (c); and the arguments in print function calls may contain function calls, as in (d).

What if a function is to return more than one value, such as function maxmin to

return *both* the maximum and minimum values of a list of integers? In Python, we can do this by returning the two values as a single tuple,

function definition
def maxmin(num_list):
return (max(num_list), min(num_list))
function use
weekly_temps 5 [45, 30, 52, 58, 62, 48, 49] (a) highlow_temps 5 maxmin(weekly_temps) (b) high, low 5 maxmin(weekly_temps)

In (a) above, the returned tuple is assigned to a single variable, highlow_temps. Thus, highlow_ temps[0] contains the maximum temperature, and highlow_temps[1] contains the minimum temperature. In (b), however, a *tuple assignment* is used. In this case, variables high and low are each assigned a value of the tuple based on the order that they appear. Thus, high is assigned to the tuple value at index 0, and low the tuple value at index 1 of the returned tuple.

Note that it does not make sense for a call to a value-returning function to be used as a statement, for example,
max(num_list)
Such a function call does not have any utility because the expression would evaluate to a value that is never used and thus is effectively “thrown away.”

Finally, we can design value-returning functions that do not take any arguments, as we saw in the getConvertTo function of the previous temperature conversion program. Empty parentheses are used in both the function header and the function call. This is needed to distinguish the identifier as denoting a function name and not a variable.

LET’S TRY IT Enter the definitions of functions avg (from section 5.1.2) and minmax given above. Then enter the following function calls and observe the results. ...avg(10,25,40)
???
... num_list 5 [10,20,30]

... avg(10,25,40) 1 10 ???
... if avg(10,25,240) , 0:

print 'Invalid avg' ???
... max_min 5 maxmin(num_list) ... max_min[0]

???
... max_min[1]
???

... avg(avg(2,4,6),8,12) ???
... avg(1,2,3) * avg(4,5,6) ???
... max, min 5 maxmin(num_list) ... max
???
... min
???

Function calls to value-returning functions can be used anywhere that a function’s return value is appropriate.

5.2.2 Calling Non-Value-Returning Functions

As we have seen, non-value-returning functions are called for their side effects, and not for a returned function value. Thus, such function calls are statements, and therefore can be used anywhere that an executable statement is allowed. Consider such a function call to displayWelcome from Figure 5-7,

displayWelcome()

It would not make sense to treat this function call as an expression, since no meaningful value is returned (only the default return value None). Thus, for example, the following assignment statement would not serve any purpose,

welcome_displayed 5 displayWelcome()

Finally, as demonstrated by function displayWelcome(), functions called for their side effects can be designed to take no arguments, the same as we saw for value-returning functions. Parentheses are still included in the function call to indicate that identifier displayWelcome is a function name, and not a variable.

LET’S TRY IT

Enter the definition of function hello given below, then enter the following function calls and observe the results.

... def sayHello(): print('Hello!')
... sayHello()
???

```
... t 5 sayHello()
???
... t
???
... t 55 None
???
... def buildHello(name):
return 'Hello' 1 name 1 '!'
... greeting 5 buildHello('Charles')
... print(greeting)
???
... buildHello('Charles')
???
... buildHello()
???
```

Function calls to non-value-returning functions can be used anywhere that an executable statement is allowed.

5.2.3 Parameter Passing

Now that we have discussed how functions are called, we take a closer look at the passing of arguments to functions.

Actual Arguments vs. Formal Parameters

Parameter passing is the process of passing arguments to a function. As we have seen, actual arguments are the values passed to a function’s formal parameters to be operated on. This is illustrated in Figure 5-8.

FIGURE 5-8 Parameter Passing

Here, the values of birthYr (the user's year of birth) and HSGradYr (the user's year of high school graduation) are passed as the actual arguments to formal parameters n1 and n2. Each call is part of the same Boolean expression ordered(birthYr, HSGradYr) and ordered(HSGradYr, colGradYr). In the second function call of the expression, a *different* set of values HSGradYr and colGradYr are passed. Formal parameter names n1 and n2, however, remain the same.

Note that the correspondence of actual arguments and formal parameters is determined by the *order* of the arguments passed, and not their names. Thus, for example, it is perfectly fine to pass an actual argument named num2 to formal parameter n1, and actual argument num1 to formal parameter n2, as given in Figure 5-9.

```
def ordered(n1, n2):
   return n1 < n2


num1 = int(input('Enter your age: '))
num2 = int(input('Enter your brother's age: '))

if ordered(num1, num2):
    print('He is your older brother')
else:
    if ordered(num2, num1):
        print('He is your younger brother')
    else:
        print('Are you twins?')
```

FIGURE 5-9

Parameter Passing and Argument Names

In this example, function ordered is called once with arguments num1, num2 and a second time with arguments num2, num1. Each is a proper function call and each is what is logically needed in this instance.

LET'S TRY IT Enter the definition of function ordered given above into the Python Shell. Then enter the following and observe the results.
... nums_1 5 [5,2,9,3] ... ordered(max(nums_1), max(nums_2)) ... nums_2 5 [8,4,6,1] ... ???
... ordered(min(nums_1), max(nums_2)) ???
The correspondence of actual arguments and formal parameters is determined by the *order* of the arguments passed, and not their names.
Mutable vs. Immutable Arguments

There is an issue related to parameter passing that we have yet to address. We know that when a function is called, the current values of the arguments passed become the initial values of their corresponding formal parameters,

def avg(n1, n2, n3):
avg(10, 25, 40)
In this case, literal values are passed as the arguments to function avg. When variables are passed as actual arguments, however, as shown below,
def avg(**n1**, **n2**, **n3**):
avg(**num1**, **num2**, **num3**)

there is the question as to whether any changes to formal parameters n1, n2, and

n3 in the function result in changes to the corresponding actual arguments num1, num2, and num3. In this case, function avg doesn’t assign values to its formal parameters, so there is no possibility of the actual arguments being changed. Consider, however, the following function,

```
def countDown(n):
while n .5 0:
if (n != 0):
print(n, '..', end='')

else:
print(n)
n 5 n 2 1
```

This function simply displays a countdown of the provided integer parameter value. For example, function call countDown(4) produces the following output,
4 . . 3 . . 2 . . 1 . . 0

What if the function call contained a variable as the argument, for example, countDown(num_ tics)? Since function countDown alters the value of formal parameter n, decrementing it until it reaches the value − 1, does the corresponding actual argument num_tics have value − 1 as well?

```
... num_tics 5 10
... countDown(num_tics) ... num_tics
???
```

If you try this, you will see that num_tics is unchanged. Now consider the following function,

```
def sumPos(nums): ... nums_1 5 [5, 22, 9, 4, 26, 1] for k in range(0, len(nums)):
... total 5 sumPos(nums_1) if nums[k] , 0: ... total
nums[k] 5 0 19
... nums_1
return sum(nums) [5,0,9,4,0,1]
```

Function sumPos returns the sum of only the positive numbers in the provided argument. It does this by first replacing all negative values in parameter nums with 0, then summing the list using built-in function sum. We see above that the corresponding actual argument nums_1 has been altered in this case, with all of the original negative values set to 0.

The reason that there was no change in integer argument num_tics above but there was in list argument nums_1 has to do with their types. Lists are mutable. Thus, arguments of type list will be altered if passed to a function that alters its value. Integers, floats, Booleans, strings, and tuples, on the other hand, are immutable. Thus, arguments of these types cannot be altered as a result of any function call.

It is generally better to design functions that do not return results through their arguments. In most cases, the result should be returned as the function's return value. What if a function needs to return more than one function value? The values can be returned in a tuple, as discussed above.

LET'S TRY IT Enter the following and observe the results. ... num 5 10

... def incr(n): n 5 n 1 1
... incr(num)
... num
??? ??? ... nums_1 5 [1,2,3] ... nums_2 5 (1,2,3) ... def update(nums): ...
update(nums_2) nums[1] 5 nums[1] 1 1 ... ???
... update(nums_1)
... nums_1

Only arguments of mutable type can be altered when passed as an argument to a function. In general, function results should be through a function's return value, and not through altered parameters.
5.2.4 Keyword Arguments in Python

The functions we have looked at so far were called with a fixed number of positional arguments. A **positional argument** is an argument that is assigned to a particular parameter based on its position in the argument list, as illustrated below.

def mortgage_rate(amount, rate, term)
monthly_payment 5 mortgage_rate(350000, 0.06, 20)
This function computes and returns the monthly mortgage payment for a given loan amount (amount), interest rate (rate), and number of years of the loan (term).

Python provides the option of calling any function by the use of keyword arguments. A **keyword argument** is an argument that is specified by parameter

name, rather than as a positional argument as shown below (note that keyword arguments, by convention, do not have a space before or after the equal sign),

def mortgage_rate(amount, rate, term)
monthly_payment 5 mortgage_rate(rate 50.06, term 520, amount 5350000)

This can be a useful way of calling a function if it is easier to remember the parameter names than it is to remember their order. It is possible to call a function with the use of both positional and keyword arguments. However, all positional arguments must come before all keyword arguments in the function call, as shown below.

def mortgage_rate(amount, rate, term)
monthly_payment 5 mortgage_rate(35000, term 520, rate 50.06)
This form of function call might be useful, for example, if you remember that the first argument is the loan amount, but you are not sure of the order of the last two arguments rate and term.
LET'S TRY IT
Enter the following function definition in the Python Shell. Execute the statements below and observe the results.
... def addup(first, last): ... addup(1,10) ???
if first . last:
sum 5 21 1, last510)

else: ???
sum 5 0
for i in range(first, last11): ... addup(last510, first51)

sum 5 sum 1 i ???
return sum
A **positional argument** is an argument that is assigned to a particular parameter based on its position in the argument list. A **keyword argument** is an argument that is specified by parameter name.
5.2.5 Default Arguments in Python

Python also provides the ability to assign a default value to any function parameter allowing for the use of default arguments. A **default argument** is an argument that can be optionally provided, as shown here.

def mortgage_rate(amount, rate, term 520)

monthly_payment 5 mortgage_rate(35000, 0.62)

In this case, the third argument in calls to function mortgage_rate is optional. If omitted, parameter term will default to the value 20 (years) as shown. If, on the other hand, a third argument is provided, the value passed replaces the default parameter value. All positional arguments must come before any default arguments in a function definition.

LET'S TRY IT Enter the following function definition in the Python Shell. Execute the statements below and observe the results.
... def addup(first, last, incr51): addup(1,10)

???
if first . last:
sum 5 21

else:
sum 5 0
for i in range(first, last11, incr):

sum 5 sum 1 i
return sum
... addup(1,10,2) ???
... addup(first=1, last=10) ???
... addup(incr=2, first=1,

last=10) ???
A **default argument** is an argument that can be optionally provided in a given function call. When not provided, the corresponding parameter provides a default value.
5.2.6 Variable Scope

Looking back at the temperature conversion program in section 5.1.3, we see that functions displayFahrenToCelsius and displayCelsiusToFahren each contain variables named temp and converted_temp. We ask, "Do these identifiers refer to common entities, or does each function have its own distinct entities?" The answer is based on the concept of identifier *scope*, which we discuss next.

Local Scope and Local Variables

A **local variable** is a variable that is only accessible from within a given function. Such variables are said to have **local scope**. In Python, any variable assigned a value in a function becomes a local variable of the function. Consider the example in Figure 5-10.

```
def func1():
    n = 10
    print('n in func1 = ', n)
```

```
def func2():
    n = 20
    print('n in func2 before call to func1 = ', n)
    func1()
    print('n in func2 after call to func1 = ', n)
```

```
>>> func2()
n in func2 before call to func1 =  20
n in func1 =  10
n in func2 after call to func1 =  20
```

FIGURE 5-10 Defining Local Variables

Both func1 and func2 contain identifier n. Function func1 assigns n to 10, while function func2 assigns n to 20. Both functions display the value of n when called —func2 displays the value of n both *before* and *after* its call to func1. If identifier n represents the same variable, then shouldn't its value change to 10 after the call to func1? However, as shown by the output, the value of n remains 20. This is because there are *two* distinct instances of variable n, each local to the function assigned in and inaccessible from the other.

Now consider the example in Figure 5-11. In this case, the functions are the same as above except that the assignment to variable n in func1 is commented out.

```
def func1():
    # n = 10
    print('n in func1 = ', n)
```

```
def func2():
    n = 20
    print('n in func2 before call to func1 = ', n)
    func1()
    print('n in func2 after call to func1 = ', n)
```

```
>>> func2()
n in func2 before call to func1 =  20
Traceback (most recent call last):
  .
  .
    print('n in func1 = ', n)
NameError: global name 'n' is not defined
```

FIGURE 5-11 Inaccessibility of Local Variables

In this case, we get an error indicating that variable n is not defined within func1. This is because variable n defined in func2 is inaccessible from func1. (In this case, n is expected to be a *global* variable, discussed next.)

The period of time that a variable exists is called its **lifetime**. Local variables are automatically created (allocated memory) when a function is called, and destroyed (deallocated) when the function terminates. Thus, the lifetime of a local variable is equal to the duration of its function's execution. Consequently, the values of local variables are not retained from one function call to the next.

The concept of a local variable is an important one in programming. It allows variables to be defined in a function without regard to the variable names used in other functions of the program. It also allows previously written functions to be easily incorporated into a program. The use of global variables, on the other hand, brings potential havoc to programs, discussed next.

LET'S TRY IT Enter the following function definition in the Python Shell. Execute the statements below and observe the results.
... def func1(): ... func1() some_var 5 10 ... some_var ???
A **local variable** is a variable that is only accessible from within the function it resides. Such variables are said to have **local scope**.

Global Variables and Global Scope

A **global variable** is a variable that is defined outside of any function definition. Such variables are said to have **global scope**. This is demonstrated in Figure 5-12.

FIGURE 5-12 Access to Value of Global Variable

V ariable max is defined outside func1 and func2 and therefore “global” to each. As a result, it is directly accessible by both functions. For this reason, *the use of global variables is generally considered to be bad programming style*. Although it provides a convenient way to share values among functions, *all* functions within the scope of a global variable can access and alter it. This may include functions that have no need to access the variable, but none-the-less may unintentionally alter it.

Another reason that the use of global variables is bad practice is related to code reuse. If a function is to be reused in another program, the function will not work properly if it is reliant on the existence of global variables that are nonexistent in the new program. Thus, it is good programming practice to design functions so all data needed for a function (other than its local variables) are explicitly passed as arguments, and not accessed through global variables.

A **global variable** is a variable defined outside of any function definition. Such variables are said to have **global scope**. The use of global variables is considered bad programming practice.

5.2.7 Let’s Apply It—GPA Calculation Program

The following program (Figure 5-14) computes a semester GPA and new cumulative GPA for a given student. This program utilizes the following programming features:

tuple assignment ➤

Figure 5-13 illustrates an example execution of the program.

```
This program calculates semester and cumulative GPAs

Enter total number of earned credits: 30
Enter your current cumulative GPA: 3.25

Enter grade (hit Enter if done): A
Enter number of credits: 4
Enter grade (hit Enter if done): A
Enter number of credits: 3
Enter grade (hit Enter if done): B
Enter number of credits: 3
Enter grade (hit Enter if done): B
Enter number of credits: 3
Enter grade (hit Enter if done): A
Enter number of credits: 3
Enter grade (hit Enter if done):

Your semester GPA is 3.62
Your new cumulative GPA is 3.38
>>>
```

FIGURE 5-13 Execution of GPA Calculation Program

```
# Semester GPA Calculation

def convertGrade(grade):
    if grade == 'F':
        return 0
    else:
        return 4 - (ord(grade) - ord('A'))

def getGrades():
    semester_info = []
    more_grades = True
    empty_str = ''

```

FIGURE 5-14 GPA Calculation Program (*Continued*)

```
14      while more_grades:
15          course_grade = input('Enter grade (hit Enter if done): ')
16          while course_grade not in ('A','B','C','D','F',empty_str):
17              course_grade = input('Enter letter grade received: ')
18              if course_grade == empty_str:
19              more_grades = False
20          else:
21              num_credits = int(input('Enter number of credits: '))
22              semester_info.append([num_credits, course_grade])
23
24  def calculateGPA(sem_grades_info, cumm_gpa_info):
25      sem_quality_pts = 0
26      sem_credits = 0
27      current_cumm_gpa, total_credits = cumm_gpa_info
28
29      for k in range(len(sem_grades_info)):
30          num_credits, letter_grade = sem_grades_info[k]
31
32          sem_quality_pts = sem_quality_pts + \
33                            num_credits * convertGrade(letter_grade)
34
35          sem_credits = sem_credits + num_credits
36
37      sem_gpa = sem_quality_pts / sem_credits
38      new_cumm_gpa = (current_cumm_gpa * total_credits + sem_gpa * \
39                      sem_credits) /(total_credits + sem_credits)
40
41      return (sem_gpa, new_cumm_gpa)
42
43  # ---- main
44
45  # program greeting
46  print('This program calculates new semester and cummulative GPAs\n')
47
48  # get current GPA info
49  total_credits = int(input('Enter total number of earned credits: '))
50  cumm_gpa = float(input('Enter your current cummulative GPA: '))
51  cumm_gpa_info = (cumm_gpa, total_credits)
52
53  # get current semester grade info
54  print()
55  semester_grades = getGrades()
56
57  # calculate semester gpa and new cummulative gpa
58  semester_gpa, cumm_gpa = calculateGPA(semester_grades, cumm_gpa_info)
59
60  # display semester gpa and new cummulative gpa
61  print('\nYour semester GPA is', format(semester_gpa, '.2f'))
62  print('Your new cummulative GPA is', format(cumm_gpa, '.2f'))
```

FIGURE 5-14 GPA Calculation Program

The program begins with the display of the program greeting on **line 49**. **Lines 49–50** get the number of earned credits (total_credits) and current cumulative

GPA (cumm_gpa) from the user. These two variables are bundled into a tuple named cumm_gpa_info on **line 51**. Since they are always used together, bundling these variables allows them to be passed to functions as one parameter rather than as separate parameters.

Function getGrades is called on **line 55**, which gets the semester grades from the user and assigns it to variable semester_grades. The value returned by function getGrades is a list of sublists, in which each sublist contains the letter grade for a given course, and the associated number of credits,

[['A', 3], ['B', 4], ['A', 3], ['C', 3]]

On **line 58**, function calculateGPA is called with arguments semester_grades and cumm_ gpa_info. The function returns a tuple containing the semester GPA and new cumulative GPA of the user. A tuple assignment is used to unpack the two values into variables semester_gpa and cumm_gpa. Finally, these values are displayed on **lines 61** and **62**.

Function calculateGPA is defined in **lines 24–41** with parameters sem_grades_ info and cumm_gpa_info. A GPA is calculated as the total quality points earned for a given set of courses, divided by the total number of credits the courses are worth. The number of *quality points* for a given course is defined as a course grade times the number of credits the course is worth. Thus, assuming a grade of A is worth 4 points, B worth 3 points, and grades of C, D, and F worth 2, 1 and 0 points, respectively, to calculate the semester GPA for a student receiving A's in two four-credit courses, B's in two three-credit courses, and a C in a one-credit course would be,

(4 * 4 1 4 * 4 1 3 * 3 1 3 * 3 1 2 * 1) / 15 5 3.47
A A B B C
where 15 is the total number of credits of all courses.

Similarly, in order to calculate a new cumulative GPA, the total quality points of the current cumulative GPA plus the total quality points of the new semester GPA is divided by the total number of credits the student has earned to date. Thus, to calculate a new cumulative GPA for a current cumulative GPA of 3.25 earning thirty credits, and a new semester GPA as given above (3.47 earning fifteen credits) would be,

(3.25 * 30 1 3.47 * 15) / 45 5 3.32

with 45 total earned credits. Thus, in function calculateGPA, local variables sem_quality_ pts and sem_credits are initialized to zero. Their values for the courses provided in parameter sem_grades_info are computed in the for loop on **lines 29–35**. This loop also calculates the semester quality points and the number of credits of the current semester, assigned to local variables sem_quality_pts and sem_credits, respectively (at **lines 32** and **35**). Note that in the calculation of the semester quality points, function convertGrade is called to convert each letter grade to its corresponding numerical value. Finally, at the end of function calculateGPA, local variable sem_gpa is assigned to the total semester quality points divided by the total semester credits. Similarly, local variable new_cumm_gpa is assigned to the total quality points to date (current_cumm_gpa * total_credits 1sem_gpa * sem_credits) divided by the total number of credits earned to date (num_credits 1sem_credits). Finally, on **line 41**, a tuple is returned containing both of these computed values.

The remaining functions defined in this program are convertGrade and getGrades. Function convertGrade is passed a letter grade, and returns the corresponding numerical value. Since the ordinal value (via the ord function) of letters in Python are sequential integers, determining the difference between the ordinal value of A and the ordinal value of a given letter grade allows the numerical value of the letter grade to be determined. For example, for a letter grade of A through D, its numerical value is determined and returned as,

return 4 2 (ord('A') 2 ord('A')) → return 4
return 4 2 (ord('B') 2 ord('A')) → return 3
return 4 2 (ord('C') 2 ord('A')) → return 2
return 4 2 (ord('D') 2 ord('A')) → return 1

Since there is no letter of grade E used, a grade of F has to be handled separately.

Finally , function getGrades returns a list of sublists of grades and credits entered by the user, as mentioned above. Thus, local variable semester_info is initialized to an empty list on **line 10**. The while loop at **line 14** iterates until Boolean variable more_grades is False, initialized to True in **line 11**. The loop continues to iterate and append another pair of grade/credits to the list until the user hits the Enter key when prompted for a course grade (**line 15**).

Self-Test Questions **1.** A function call can be made anywhere within a program in which the return type of the function is appropriate. (TRUE/FALSE)

2. An expression may contain more than one function call. (TRUE/FALSE)
3. Function calls may contain arguments that are function calls. (TRUE/FALSE)
4. All value-returning functions must contain at least one parameter. (TRUE/FALSE)
5. Every function must have at least one mutable parameter. (TRUE/FALSE)

6. A local variable in Python is a variable that is,
(a) defined inside of every function in a given program
(b) local to a given program
(c) only accessible from within the function it is defined

7. A global variable is a variable that is defined outside of any function definition. (TRUE/FALSE)
8. The use of global variables is a good way to allow different functions to access and modify the same variables. (TRUE/FALSE)

ANSWERS: 1. True, 2. True, 3. True, 4. False, 5. False, 6. (c), 7. True, 8. False

COMPUTATIONAL PROBLEM SOLVING

5.3 Credit Card Calculation Program

In this section, we design, implement, and test a program that will allow us to determine the length of time needed to pay off a credit card balance, as well as the total interest paid.

5.3.1 The Problem

The problem is to generate a table showing the decreasing balance and accumulating interest paid on a credit card account for a given credit card balance, interest rate, and monthly payment, as shown in Figure 5-15.

Year	Balance	Interest Paid
1	330.18	5.17
	315.13	10.13
	299.85	14.85
	284.35	19.35
	268.62	23.62
	252.65	27.65
	236.44	31.44
	219.98	34.98
	203.28	38.28
	186.33	41.33
	169.13	44.13
2	151.66	46.66
	133.94	48.94
	115.95	50.95
	97.69	52.69
	79.15	54.15
	60.34	55.34
	41.25	56.25
	21.86	56.86
	2.19	57.19
	0.00	57.22

FIGURE 5-15 Example Execution of the Credit Card Calculation Program

5.3.2 Problem Analysis

The factors that determine how quickly a loan is paid off are the amount of the loan, the interest rate charged, and the monthly payments made. For a fixed-rate home mortgage, the monthly payments are predetermined so that the loan is paid off within a specific number of years. Therefore, the total interest that will be paid on the loan is made evident at the time the loan is signed.

For a credit card, there is only a minimum payment required each month. It is not always explicitly stated by the credit card company, however, how long it would take to pay off the card by making only the minimum payment. The minimum payment for a credit card is dependent on the particular credit card

company. However, it is usually around 2–3% of the outstanding loan amount each month, and no less than twenty dollars. Thus, calculating this allows us to project the amount of time that it would take before the account balance becomes zero, as well as the total interest paid.

5.3.3 Program Design

Meeting the Program Requirements

No particular format is specified for how the output is to be displayed. All that is required is that the user be able to enter the relevant information and that the length of time to pay off the loan and the total interest paid is displayed. The user will also be given the choice of assuming the monthly payment to be the required minimum payment, or a larger specified amount.

Data Description

All that needs to be represented in this program are numerical values for the loan amount, the interest rate, and the monthly payment made. There is no need to create a data structure as the table of payments can be generated as it is displayed.

Algorithmic Approach

The only algorithm needed for this problem is the calculation of the required minimum payment. The minimum payment is usually calculated at 2% or 3% of the outstanding balance, with a lower limit of around $20. Therefore, we will assume a worst case scenario of a minimum payment calculated at 2%, with a minimum payment of $20.

Overall Program Steps

The overall steps in this program design are given in Figure 5-16.

FIGURE 5-16 Overall Steps of the Credit Card Calculation Program

5.3.4 Program Implementation and Testing
Stage 1—Developing the Overall Program Structure

We first develop and test the overall program structure given in Figure 5-17.

The program begins on **line 15** with a call to function displayWelcome(). Next, the current credit card balance and annual interest rate (APR) are input from the user (**lines 18–19**), each read as an integer value. Since the monthly interest rate is what will be used in the calculations, the value in apr is divided by 1200 (on **line 21**). This converts the value to a monthly interest rate, as well as converting it to decimal form (for example, 18% as 0.18).

The final value input from the user is the monthly payments that they wish to have the payoff calculated with. They have a choice of either going with the minimum required monthly payment, assumed to be $20 for testing purposes (**line 28**), or a specified monthly payment (**line 31**). The credit card balance, annual percentage rate, and the assumed monthly payments are passed to function displayPayments (on **line 34**) to calculate and display the pay down of the balance as well as the interest paid over each month of the payoff period.

```
1  # Credit Card Calculation Program (Stage 1)
2
3  def displayWelcome():
4      print('\n.... Entering function display welcome')
5
6  def displayPayments(balance, int_rate, monthly_payment):
7      print('\n.... Entering function displayPayments')
8      print('parameter balance =', balance)
9      print('parameter int_rate =', int_rate)
10     print('parameter monthly_payment =', monthly_payment)
11
12 # ---- main
13
14 # display welcome screen
15 displayWelcome()
16
17 # get current balance and APR
18 balance = int(input('\nEnter the balance on your credit card: '))
19 apr = int(input('Enter the interest rate (APR) on the card: '))
20
21 monthly_int_rate = apr/1200
22
23 # determine monthly payment
24 response = input('Use the minimum monthly payment? (y/n): ')
25
26 if response in ('y','Y'):
27     print('Minimum payment selected')
28     monthly_payment = 20
29 else:
30     print('User-entered monthly payments selected')
31     monthly_payment = input('Enter monthly payment: ')
32
33 # display monthly payoff
34 displayPayments(balance, monthly_int_rate, monthly_payment)
```

FIGURE 5-17 Credit Card Calculation Program (Stage 1)

Functions displayWelcome (**line 3**) and displayPayments (**line 6**) consist only of **trace statements**. A trace statement prints, for testing purposes, a message indicating that a certain point in the program has been reached. Trace statements are also used to display the value of certain variables. Once this part of the program is working, we can focus on implementing the functions and further developing the main program section.

Stage 1 Testing
We show sample test runs of this version of the program in Figure 5-18.

From the test results, we see that the appropriate values are being input and passed to function displayPayments. So it looks like the overall structure of this

stage of the program is working correctly.

Stage 2—Generating an Unformatted Display of Payments

We next implement function displayWelcome, and develop an initial implementation of function displayPayments,given in Figure 5-19. We remove the two print instructions that were included only for test purposes in stage 1 of the program (previously on **lines 27** and **30**).

```
.... Entering function display welcome

Enter the balance on your credit card: 1500
Enter the interest rate (APR) on the card: 18
Use the minimum monthly payment? (y/n): n
User-entered monthly payments selected
Enter monthly payment: 100

.... Entering function displayPayments
parameter balance = 1500
parameter int_rate = 0.015
parameter monthly_payment = 100
>>>
```

```
.... Entering function display welcome

Enter the balance on your credit card: 1500
Enter the interest rate (APR) on the card: 18
Use the minimum monthly payment? (y/n): y
Minimum payment selected

.... Entering function displayPayments
parameter balance = 1500
parameter int_rate = 0.015
parameter monthly_payment = 20
>>>
```

FIGURE 5-18 Output of First Stage Testing

Also, the minimum required monthly payment is computed (**lines 45–48**) rather than being set to 20.

Function displayPayments is where most of the work is done in the program. Therefore, we shall develop this function in stages as well. At this point, we develop the function to display, for each month during the loan payoff, the year, the current balance, and the total interest paid to date. We delay issues of screen formatting for the alignment of numbers, and only include formatting for

rounding numeric values to two decimal places.

The while loop on **line 21** iterates while balance, passed as an argument to the function, is greater than zero. The function will keep count of the number of months (lines) displayed, as well as the total interest paid. Variables num_months and total_int_paid are used for this purpose, and are therefore initialized before the loop to 0 (**lines 11–12**). On **lines 15–18** the initial information for the calculation is displayed. Within the while loop, on **line 22**, the monthly interest paid (monthly_int) is computed as the current balance of that month during the payoff period (balance), times the monthly interest rate (int_rate). The total interest paid is then updated on **line 23**. On **line 24**, the new balance is computed as the current balance, plus the interest for the month, minus the monthly payment.

The next step is to display these computed values. Since time is kept track of in terms of months, the current year to be displayed is computed using integer division (**line 26**), adding one so that the first year is displayed as 1, and not 0. Then on **line 27**, the line representing the payment for the current month is displayed. Formatting is used so that all numerical values are displayed with two decimal places. Finally, variable num_months is incremented by one for the next iteration of the loop.

Stage 2 Testing

We test this program once for a specified monthly payment amount, and once for the option of minimum monthly payments. The results are given in Figures 5-20 and 5-21.

```python
# Credit Card Calculation Program (Stage 2)

def displayWelcome():
    print('This program will determine the time to pay off a credit')
    print('card and the interest paid based on the current balance,')
    print('the interest rate, and the monthly payments made.')

def displayPayments(balance, int_rate, monthly_payment):

    # init
    num_months = 0
    total_int_paid = 0

    # display loan info
    print('PAYOFF SCHEDULE')
    print('\nCredit card balance: $' + format(balance,'.2f'))
    print('Annual interest rate:', str(1200 * int_rate)+'%')
    print('Monthly payment: $', format(monthly_payment,'.2f'))

    # display year-by-year account status
    while balance > 0:
        monthly_int = balance * int_rate
        total_int_paid = total_int_paid + monthly_int
        balance = balance + monthly_int - monthly_payment

        year = (num_months // 12) + 1
        print(year, format(balance,'.2f'), format(total_int_paid,'.2f'))

        num_months = num_months + 1

# ---- main

# display welcome screen
displayWelcome()

# determine current balance and APR
balance = int(input('\nEnter the balance on your credit card: '))
apr = int(input('Enter the interest rate (APR) on the card: '))

monthly_int_rate = apr/1200

# determine monthly payment
response = input('Use the minimum monthly payment? (y/n): ')

if response in ('y','Y'):
    if balance < 1000:
        monthly_payment = 20
    else:
        monthly_payment = balance * .02
else:
    monthly_payment = input('Enter monthly payment: ')

# display monthly payoff
displayPayments(balance, monthly_int_rate, monthly_payment)
```

FIGURE 5-19 Credit Card Calculation Program (Stage 2)

```
This program will determine the time to pay off a credit
card and the interest paid based on the current balance,
the interest rate, and the monthly payments made.

Enter the balance on your credit card: 1500
Enter the annual interest rate (APR) on the card: 18
Use the minimum monthly payment? (y/n): n
Enter monthly payment: 100
PAYOFF SCHEDULE

Credit card balance: $1500.00
Annual interest rate: 18.0%
Traceback (most recent call last):
  File "C:\My Python Programs\CreditCardCalc-Stage2 with
error.py", line 53, in <module>
    displayPayments(balance, monthly_int_rate, monthly_payment)
  File "C:\My Python Programs\CreditCardCalc-Stage2 with
error.py", line 18, in displayPayments
    print('Monthly payment: $', format(monthly_payment,'.2f'))
ValueError: Unknown format code 'f' for object of type 'str'
>>>
```

FIGURE 5-20 Output of Second Stage Testing (User-Entered Payment)

```
This program will determine the time to pay off a credit
card and the interest paid based on the current balance,
the interest rate, and the monthly payments made.

Enter the balance on your credit card: 350
Enter the annual interest rate (APR) on the card: 18
Use the minimum monthly payment? (y/n): y
PAYOFF SCHEDULE

Credit card balance: $350.00
Annual interest rate: 18.0%
Monthly payment: $ 20.00
1 335.25 5.25
1 320.28 10.28
1 305.08 15.08
1 289.66 19.66
1 274.00 24.00
1 258.11 28.11
1 241.99 31.99
1 225.62 35.62
1 209.00 39.00
1 192.13 42.13
1 175.02 45.02
1 157.64 47.64
2 140.01 50.01
2 122.11 52.11
2 103.94 53.94
2 85.50 55.50
2 66.78 56.78
2 47.78 57.78
2 28.50 58.50
2 8.93 58.93
2 -10.94 59.06
>>>
```

FIGURE 5-21 Output of Second Stage Testing (Minimum Payment)

Clearly, there is something wrong with this version of the program. The ValueError generated in Figure 5-20 indicates that the format specifier .2f is an unknown format code for a string type value, referring to **line 18**. Thus, this must be referring to variable monthly_payment. But that should be a numeric value, and not a string value! How could it have become a string type? Let's check if the problem also occurs when selecting the minimum payment option (Figure 5-21).

In this case the program works. Since the problem only occurred when the user entered the monthly payment (as opposed to the minimum payment option), we

next try to determine what differences there are in the program related to the assignment of variable monthly_payment.

```
# determine monthly payment
response 5 input('Use the minimum monthly payment? (y/n): ') if response in
('y', 'Y'):

if balance < 1000:
monthly_payment 5 20
else:
monthly_payment = balance * .02
else:
monthly_payment = input('Enter monthly payment: ')
```

When the user selects the minimum monthly payment option, variable monthly_payment is set to integer value 20 (or 2% of the current balance if balance is greater than 1000). Otherwise, its value is input from the user. This variable is not redefined anywhere else in the program. Since the variable monthly_payment is not a local variable, we can display its value directly from the Python shell,

```
... monthly_payment '140'
```

It *is* a string value. We immediately realize that the input value for variable monthly_payment was not converted to an integer type, and was thus left as a string type! We fix this problem by replacing the line with the following,

monthly_payment 5 **int(**input('Enter monthly payment: ')

This explains why the problem did not appear in the testing of stage 1 of the program. In that version, variable monthly_payment was never formatted as a numeric value, and also never used in a numerical calculation (both of which would have generated an error).

At this point, we execute a number of test cases for various initial balances, interest rates, and monthly payments. The result is given in Figure 5-22, checked against online loan payoff calculator tools. We next move on to the final stage of program development.

Stage 3—Formatting the Displayed Output

In this final stage of the program, input error checking is added. The program is also modified to allow the user to continue to enter various monthly payments for recalculating a given balance payoff. Output formatting is added to make the displayed information more readable. Finally, we correct the display of a negative balance at the end of the payoff schedule, as appears in Figure 5-21. The final version of the program is given in Figure 5-23.

Payoff Information			**Expected Results**		**Actual Results**		**Evaluation**
Balance	Interest Rate	Monthly Payment	Num Months	Interest Paid	Num Months	Interest Paid	
250	18%	$20 (min)	14	28.93	14	28.93	Passed
600	14%	$20 (min)	38	142.80	38	142.80	Passed
12,000	20%	$240 (min)	109	14,016.23	109	14,016.23	Passed
250	18%	$40	7	14.54	7	14.54	Passed
600	14%	$50	14	50.15	14	50.15	Passed
12,000	20%	$400	42	4,773.98	42	4,773.98	Passed

FIGURE 5-22 Test Cases for Stage 2 of the Credit Card Calculation Program

```python
# Credit Card Calculation Program (Final Version)

def displayWelcome():
    print('This program will determine the time to pay off a credit')
    print('card and the interest paid based on the current balance,')
    print('the interest rate, and the monthly payments made.')

def displayPayments(balance, int_rate, monthly_payment):

    # init
    num_months = 0
    total_int_paid = 0
    payment_num = 1

    empty_year_field = format(' ', '8')

    # display heading
    print('\n', format('PAYOFF SCHEDULE','>20'))
    print(format('Year','>10') + format('Balance','>10') +
          format('Payment Num', '>14') + format('Interest Paid','>16'))

    # display year-by-year account status
    while balance > 0:
        monthly_int = balance * int_rate
        total_int_paid = total_int_paid + monthly_int

        balance = balance + monthly_int - monthly_payment

        if balance < 0:
            balance = 0

        if num_months % 12 == 0:
            year_field = format(num_months // 12 + 1, '>8')
        else:
            year_field = empty_year_field

        print(year_field + format(balance, '>12,.2f') +
              format(payment_num, '>9') +
              format(total_int_paid, '>17,.2f'))

        payment_num = payment_num + 1
        num_months = num_months + 1

```

```
44 # ---- main
45
46 # display welcome screen
47 displayWelcome()
48
49 # get current balance and APR
50 balance = int(input('\nEnter the balance on your credit card: '))
51 apr = int(input('Enter the interest rate (APR) on the card: '))
52
53 monthly_int_rate = apr/1200
54
55 yes_response = ('y','Y')
56 no_response = ('n','N')
57
58 calc = True
59 while calc:
60
61     # calc minimum monthly payment
62     if balance < 1000:
63         min_monthly_payment = 20
64     else:
65         min_monthly_payment = balance * .02
66
67     # get monthly payment
68     print('\nAssuming a minimum payment of 2% of the balance ($20 min)')
69     print('Your minimum payment would be',
70           format(min_monthly_payment, '.2f'),'\n')
71
72     response = input('Use the minimum monthly payment? (y/n): ')
73     while response not in yes_response + no_response:
74         response = input('Use the minimum monthly payment? (y/n): ')
75
76     if response in yes_response:
77         monthly_payment = min_monthly_payment
78     else:
79         acceptable_payment = False
80
81         while not acceptable_payment:
82             monthly_payment = int(input('\nEnter monthly payment: '))
83
84             if monthly_payment < balance * .02:
85                 print('Minimum payment of 2% of balance required ($' +
86                       str(balance * .02) + ')')
87
88             elif monthly_payment < 20:
89                 print('Minimum payment of $20 required')
90             else:
91                 acceptable_payment = True
92
```

```
93      # check if single payment pays off balance
94      if monthly_payment >= balance:
95          print('* This payment amount would pay off your balance *')
96      else:
97          # display month-by-month balance payoff
98          displayPayments(balance, monthly_int_rate, monthly_payment)
99
100         # calculate again with another monthly payment?
101         again = input('\nRecalculate with another payment? (y/n): ')
102         while again not in yes_response + no_response:
103             again = input('Recalculate with another payment? (y/n): ')
104
105         if again in yes_response:
106             calc = True    # continue program
107             print('\n\nFor your current balance of $' + str(balance))
108         else:
109             calc = False   # terminate program
```

FIGURE 5-23 Final Stage of the Credit Card Calculation Program

The first set of changes in the program provides some input error checking. (We will address means of more complete error checking in Chapter 7.) In **lines 55–56**, tuples yes_response and no_ response are defined. These are used to check if input from the user is an appropriate yes/no response. For example, the while statement on **line 73** checks that the input from **line 72** is either 'y', 'Y' ,'n', or 'N',

while response not in yes_response 1 no_response:
by checking if response is in the concatenation of tuples yes_response and no_response. For determining the specific response, the tuples can be used individually (**line 76**),
if response in yes_response:
Similar input error checking is done on **line 101**.

The next set of changes allows a number of payoff schedules for an entered balance to be calculated. A while statement is added at **line 59**, with its condition based on the value of Boolean variable calc (initialized to True on **line 58**). To accommodate the recalculation of payoff schedules, variables num_months, total_int_paid and payment_num are each reset to 0 in function displayPayments (**lines 11–13**).

Output formatting is added in function displayPayments. On **line 18**, 'PAYOFF SCHEDULE' is displayed right-justified within a field of twenty. On **lines 19–20**, the column headings are displayed with appropriate field widths. **Lines 37–39** display the balance, payment number and interest of each month, aligned

under the column headings. **Lines 32–35** ensure that each year is displayed only once. Finally, in **lines 29–30** variable balance is set to zero if it becomes negative so that negative balances are not displayed.

Stage 3 Testing

We give example output of this version of the program for both a payoff using the required minimum monthly payment, and for a user-entered monthly payment in Figures 5-24 and 5-25.

Figure 5-25 depicts a portion of the output for the sake of space. We run the same set of test cases used in the testing of the previous (stage 2) version of the program, given in Figure 5-26.

Based on these results, we can assume that the program is functioning properly.

```
This program will determine the time to pay off a credit
card and the interest paid based on the current balance,
the interest rate, and the monthly payments made.

Enter the balance on your credit card: 1500
Enter the interest rate (APR) on the card: 18

Assuming a minimum payment of 2% of the balance ($20 min)
Your minimum payment would be 30.00

Use the minimum monthly payment? (y/n): n

Enter monthly payment: 100

      PAYOFF SCHEDULE
      Year   Balance   Payment Num   Interest Paid
       1    1,422.50        1            22.50
            1,343.84        2            43.84
            1,264.00        3            64.00
            1,182.95        4            82.95
            1,100.70        5           100.70
            1,017.21        6           117.21
              932.47        7           132.47
              846.45        8           146.45
              759.15        9           159.15
              670.54       10           170.54
              580.60       11           180.60
              489.31       12           189.31
       2      396.65       13           196.65
              302.60       14           202.60
              207.13       15           207.13
              110.24       16           210.24
               11.89       17           211.89
                0.00       18           212.07

Recalculate with another payment? (y/n): n
>>>
```

FIGURE 5-24 Output of Third Stage Testing (User-Entered Payment)

```
This program will determine the time to pay off a credit
card and the interest paid based on the current balance,
the interest rate, and the monthly payments made.

Enter the balance on your credit card: 1500
Enter the interest rate (APR) on the card: 18

Assuming a minimum payment of 2% of the balance ($20 min)
Your minimum payment would be 30.00

Use the minimum monthly payment? (y/n): y

      PAYOFF SCHEDULE
      Year   Balance   Payment Num   Interest Paid
       1    1,492.50        1            22.50
            1,484.89        2            44.89
            1,477.16        3            67.16
            1,469.32        4            89.32
                .          .           .
                .          .           .
       7      517.51       73         1,207.51
              495.28       74         1,215.28
              472.70       75         1,222.70
              449.79       76         1,229.79
                .          .           .
                .          .           .
       8      227.51       85         1,277.51
              200.92       86         1,280.92
              173.94       87         1,283.94
              146.55       88         1,286.55
              118.74       89         1,288.74
               90.53       90         1,290.53
               61.88       91         1,291.88
               32.81       92         1,292.81
                3.30       93         1,293.30
                0.00       94         1,293.35

Recalculate with another payment? (y/n): n
>>>
```

FIGURE 5-25 Output of Third Stage Testing (Minimum Payment)

Payoff Information			Expected Results		Actual Results		Evaluation
Balance	Interest Rate	Monthly Payment	Num Months	Interest Paid	Num Months	Interest Paid	
250	18%	$20 (min)	14	28.93	14	28.93	Passed
600	14%	$20 (min)	38	142.80	38	142.80	Passed
12,000	20%	$240 (min)	109	14,016.23	109	14,016.23	Passed
250	18%	$40	7	14.54	7	14.54	Passed
600	14%	$50	14	50.15	14	50.15	Passed
12,000	20%	$400	42	4,773.98	42	4,773.98	Passed

FIGURE 5-26 Test Cases for Stage 3 of the Credit Card Calculation Program

CHAPTER SUMMARY
General Topics

Program Routines
Value-Returning vs. Non-Value-Returning Functions Side Effects (of Function Calls)
Parameter Passing: Actual Arguments vs.

Formal Parameters
Local Scope and Local Variables
Global Scope and Global Variables
Variable Lifetime
Python-Specific Programming Topics

Defining Functions in Python
Built-in Functions of Python
Value-Returning and Non-Value-Returning

Functions in Python
Tuple Assignment in Python
Mutable vs. Immutable Arguments in Python Local vs. Global Variables in Python

CHAPTER EXERCISES Section 5.1

1. Function avg returns the average of three values, as given in the chapter.

Which of the following statements, each making calls to function avg, are valid? (Assume that all variables are of numeric type.) **(a)** result 5 avg(n1, n2)
(b) result 5 avg(n1, n2, avg(n3, n4, n5))
(c) result 5 avg(n1 1 n2, n3 1 n4, n5 1 n6)
(d)print(avg(n1, n2, n3))
(e) avg(n1, n2, n3)

2. Which of the following statements, each involving calls to function displayWelcome displaying a welcome message on the screen as given in the chapter, are valid?
(a) print(displayWelcome)
(b) displayWelcome
(c) result 5 displayWelcome()
(d) displayWelcome()

Section 5.2
3. Suppose there are nine variables, each holding an integer value as shown below, for which the average of the largest value in each line of variables is to be computed.

num1 5 10 num2 5 20 num3 5 25 max1 5 25
num4 5 5 num5 5 15 num6 5 35 max2 5 35
num7 5 20 num8 5 30 num9 5 25 max3 5 30
average 5 (max1 1 max2 1 max3) / 3.0
5 (25 1 35 1 30) / 3.0
5 30.0

Using functions avg and max, give an expression that computes the average as shown above. **4.** Assume that there exists a Boolean function named isLeapYear that determines if a given year is a
leap year or not. Give an appropriate if statement that prints “Year is a Leap Year” if the year passed is a
leap year, and “Year is Not a Leap Year” otherwise, for variable year.

Python Programming Exercises 203
5. For the following function definition and associated function calls,

def somefunction(n1, n2): .
.

main
num1 5 10
somefunction(num1, 15)

(a) List all the formal parameters.
(b) List all the actual arguments.
6. For the following function, indicate whether each function call is proper or not. If improper, explain why.

def gcd(n1, n2): function gcd calculates the greatest common divisor of n1 and n2, with the requirement that n1 be less than or equal to n2, and n1 and n2 are integer values.

(a) a 5 10
b 5 20
result 5 gcd(a, b)

(b) a 5 10.0
b 5 20
result 5 gcd(a, b)

(c) a 5 20
b 5 10
result 5 gcd(b, a)
(d) a 5 10
b 5 20
c 5 30
result 5 gcd(gcd(a, b), c) **(e)** a 5 10
b 5 20
c 5 30
print(gcd(a, gcd(c, b)))

PYTHON PROGRAMMING EXERCISES

P1. Write a Python function named zeroCheck that is given three integers, and returns true if any of the integers is 0, otherwise it returns false.
P2. Write a Python function named ordered3 that is passed three integers, and returns true if the three integers are in order from smallest to largest, otherwise it returns false.
P3. Write a Python function named modCount that is given a positive integer, n,

and a second positive integer, m <= n, and returns how many numbers between 1 and n are evenly divisible by m.
P4. Write a Python function named helloWorld that displays "Hello World, my name is *name*", for any given name passed to the routine.
P5. Write a Python function named printAsterisks that is passed a positive integer value n, and prints out a line of n asterisks. If n is greater than 75, then only 75 asterisks should be displayed.
P6. Write a Python function named getContinue that displays to the user “Do you want to continue (y/n): ”, and continues to prompt the user until either uppercase or lowercase 'y' or 'n' is entered, returning (lowercase) 'y' or 'n' as the function value.
P7. Implement a Python function that is passed a list of numeric values and a particular threshold value, and returns the list with all values above the given threshold value set to 0. The list should be altered as a side effect to the function call, and not by function return value.
P8. Implement the Python function described in question P7 so that the altered list is returned as a function value, rather than by side effect.

PROGRAM MODIFICATION PROBLEMS

M1. Temperature Conversion Program: Adding Kelvin Scale
Modify the Temperature Conversion program in section 5.1.3 so that it allows the user to select temperature conversion to include degrees Kelvin, in addition to degrees Fahrenheit and degrees Celsius. Include input error checking for inappropriate temperature values. (NOTE: Refer to questions M1 and M2 from Chapter 3.)

M2. GPA Calculation Program: Accommodating First-Semester Students
Modify the GPA Calculation program in section 5.2.7 so that it asks the student if this is their first semester. If so, the program should only prompt for their current semester grades, and not their cumulative GPA and total earned credits, and display their semester GPA and cumulative GPA accordingly.

M3. GPA Calculation Program: Allowing for Plus/Minus Grading
Modify the GPA Calculation program in section 5.2.7 so that it is capable of calculating a GPA for plus/ minus letter grades: A, A2, B 1, B, B2, and so forth.

M4. Credit Card Calculation Program: Summarized Output
Modify the Credit Card Calculation program in section 5.3 so that the user is

given the option of either displaying the balance and interest paid month-by-month as currently written, or to simply have the total number of months and the total interest paid without the month-by-month details.

M5. Credit Card Calculation Program: Adjustable Minimum Payment
Modify the Credit Card Calculation program in section 5.3 so that the user can enter the percentage from which the minimum monthly payment is calculated. Also modify the program so that this minimum payment percentage is displayed along with the other credit card related information.

M6. Credit Card Calculation Program: Recalculation with New Balance
Modify the Credit Card Calculation program in section 5.3 so that the program will allow the user to recalculate a new payoff schedule for a new entered balance.

PROGRAM DEVELOPMENT PROBLEMS

D1. Metric Conversion Program
Develop and test a Python program that allows the user to convert between the metric measurements of millimeter, centimeter, meter, kilometer, and inches, feet, yards, and miles. The program should be written so that any one measurement can be converted to the other.

D2. GPA Projection Program
Develop and test a Python program that lets the user enter their current cumulative GPA, their total credits earned, and the number of credits they are currently taking. The program should then request from the

user a target cumulative GPA that they wish to achieve, and display the GPA of the current semester needed to achieve it.

D3. Tic-Tac-Toe Two-Player Program
Develop and test a Python program that lets two players play tic-tac-toe. Let player 1 be X and player 2 be O. Devise a method for each player to indicate where they wish to place their symbol. The program should terminate if either there is a winner, or if the game results in a tie. The tic-tac-toe board should be displayed after every move as shown below.

```
X  ---  X
O  O  --
X  X  O
```

D4. Tic-Tac-Toe Automated Play
Develop and test a Python program that plays tic-tac-toe against the user. Develop an appropriate strategy of play and implement it in your program. The program should be designed to allow the user to continue to play new games until they decide to quit. The program should display the total number of wins by the computer versus the player at the start of each new game.

CHAPTER 6 Objects and Their Use

In procedural programming, functions are the primary building blocks of program design. In object-oriented programming, objects are the fundamental building blocks in which functions (methods) are a component. We first look at the use of individual software objects in this chapter, and in Chapter 10 look at the use of objects in object-oriented design.

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Explain the concept of an object
♦ Explain the difference between a reference and dereferenced value ♦ Describe the use of object references
♦ Explain the concept of memory allocation and deallocation ♦ Describe automatic garbage collection
♦ Explain the fundamental features of turtle graphics
♦ Effectively use objects in Python
♦ Develop simple turtle graphics programs in Python

CHAPTER CONTENTS
Motivation
Fundamental Concepts
6.1 Software Objects
6.2 Turtle Graphics
Computational Problem Solving 6.3 Horse Race Simulation Program

MOTIVATION

An object is one of the first concepts that a baby understands during its development. They understand an object as something that has a set of *attributes* (“big,” “red” ball) and a related set of *behaviors* (it rolls, it bounces).

The idea of incorporating “objects” into a programming language came out of work in computer simulation. Given the prevalence of objects in the world, it was natural to provide the corresponding notion of an object within a simulation program.

In the early 1970s, Alan Kay at Xerox PARC (Palo Alto Research Center) fully evolved the notion of object-oriented programming with the development of a programming language called Smalltalk. The language became the inspiration for the development of graphical user interfaces (GUIs)—the primary means of interacting with computers today. Before that, all interaction was through typed text. In fact, it was a visit to Xerox PARC by Steve Jobs of Apple Computers that led to the development of the first commercially successful GUI-based computer, the Apple Macintosh in 1984. Figure 6-1 lists some of the most commonly used programming languages and whether they support procedural (imperative) programming, object-oriented programming, or both. In this chapter, we look at the creation and use of objects in Python.

Programming Language	Programming Paradigm Supported	
	Procedural	Object-oriented
C (early 1970s)	X	
Smalltalk (1980)		X
C++ (mid 1980s)	X	X
Python (early 1990s)	X	X
Java (1995)		X
Ruby (mid 1990s)	X	X
C # (2000)	X	X

FIGURE 6-1
Common Programming Languages Supporting Procedural and/or Object-Oriented Programming

FUNDAMENTAL CONCEPTS

6.1 Software Objects

Objects are the fundamental component of object-oriented programming. Although we have not yet stated it, *all* values in Python are represented as objects. This includes, for example, lists, as well as numeric values. We discuss object-oriented programming in Chapter 10. In this chapter, we discuss what objects are and how they are used.

6.1.1 What Is an Object?

The notion of software objects derives from objects in the real world. All objects have certain *attributes* and *behavior*. The attributes of a car, for example, include its color, number of miles driven, current location, and so on. Its behaviors include driving the car (changing the number of miles driven attribute) and painting the car (changing its color attribute), for example.

Similarly, an **object** contains a set of attributes, stored in a set of **instance variables**, and a set of functions called **methods** that provide its behavior. For example, when sorting a list in procedural programming, there are two distinct entities—a sort function and a list to pass it, as depicted in Figure 6-2.

FIGURE 6-2
Procedural Programming Approach

In object-oriented programming, the sort routine would be *part of* the object containing the list, depicted in Figure 6-3.

FIGURE 6-3 Object names_list

Here, names_list is an object instance of the Python built-in list type. All list objects contain the same set of methods. Thus, names_list is sorted by simply calling that object's sort method, names_list.sort()

The period is referred to as the *dot operator*, used to select a member of a given object—in this case, the sort method. Note that no arguments are passed to sort. That is because methods operate on the data of the object that they are part of. Thus, the sort method does not need to be told which list to sort.

Suppose there were another list object called part_numbers, containing a list of automobile part numbers. Since all list objects behave the same, part_numbers would contain the identical set of methods as names_list. The data that they would operate on, however, would be different. Thus, two objects of the same type differ only in the particular set of values that each holds. This is depicted in Figure 6-4.

FIGURE 6-4 Object part_numbers

In order to sort *this* list, therefore, the sort method of object part_numbers is called, part_numbers.sort()

The sort routine is the same as the sort routine of object names_list. In this case, however, the list of part numbers is sorted instead. Methods append, insert, remove, count, and reverse also provide additional functionality for lists, as was discussed in Chapter 4. We next discuss the way that objects are represented in Python.

An **object** contains a set of attributes, stored in a set of **instance variables**, and a set of functions called **methods** that provide its behavior.

6.1.2 Object References

In this section we look at how objects are represented (which all values in Python are), and the effect it has on the operations of assignment and comparison, as well as parameter passing. References in Python

In Python, objects are represented as a *reference* to an object in memory, as shown in Figure 6-5.

FIGURE 6-5 Object Reference

A **reference** is a value that references, or “points to,” the location of another entity. Thus, when a new object in Python is created, *two* entities are stored—the object, and a variable holding a reference to the object. All access to the object is through the reference value. This is depicted in Figure 6-6.

FIGURE 6-6 Object References to Python Values

The value that a reference points to is called the **dereferenced value**. This is the value that the variable represents, as shown in Figure 6-7.

FIGURE 6-7 Variables’ Dereferenced Values

We can get the reference value of a variable (that is, the location in which the corresponding object is stored) by use of **built-in function id**.

...id(n) id(k) id(s) 505498136 505498136 505498296

We see that the dereferenced values of n and k, 10, is stored in the same memory location (505498136), whereas the dereferenced value of s, 20,is stored in a different location (505498296). Even though n and k are each separately assigned literal value 10, they reference the *same instance* of 10 in memory (505498136). We would expect there to be separate instances of 10 stored. Python is using a little cleverness here. Since integer values are immutable, it assigned both n and k to the same instance. This saves memory and reduces the number of reference locations that Python must maintain. From the programmer’s perspective however, they can be treated as if they are separate instances.

LET’S TRY IT From the Python Shell, first enter the following and observe the results. ... n 5 10 ... n 5 20 ... k 5 20 ... k 5 20

... id(n) ... id(n) ??? ??? ... id(k) ... id(k) ??? ???

A **reference** is a value that references, or “points to,” the location of another entity. The value that a reference points to is called the **dereferenced value**. A variable’s reference value can be determined with **built-in function** id.

The Assignment of References
With our current understanding of references, consider what happens when variable n is assigned to variable k, depicted in Figure 6-8.

FIGURE 6-8
The Assignment of References

When variable n is assigned to k, it is the *reference value*of k that is assigned, not the dereferenced value 20, as shown in Figure 6-8. This can be determined by use of the built-in id function, as demonstrated below.

... id(k) ... id(k) 55 id(n) 505498136 True
... id(n) ... n is k 505498136 True

Thus, to verify that two variables refer to the same object instance, we can either compare the two id values by use of the comparison operator, or make use of the provided is operator (which performs id(k) 55id(n)).

Thus, both n and k reference the same instance of literal value 20. This occurred in the above example when n and k were *separately* assigned 20 because integers are an immutable type, and Python makes attempts to save memory. In this case, however, n and k reference the same instance of 20 because assignment in Python assigns reference values. We must be aware of the fact, therefore, that when assigning variables referencing mutable values, such as lists, both variables reference the same list instance as well. We will discuss the implication of this next.

Finally, we look at what happens when the value of one of the two variables n or k is changed, as depicted in Figure 6-9.

FIGURE 6-9 Reassignment of Reference Value

Here, variable k is assigned a reference value to a *new* memory location holding the value 30. The previous memory location that variable k referenced is retained since variable n is still referencing it. As a result, n and k point to different values, and therefore are no longer equal. LET'S TRY IT

From the Python Shell, first enter the following and observe the results.

... k 5 10 ... k 5 30 ... n 5 k ... id(k) ... id(k) ???
??? ... id(n) ... id(n) ???
??? ... id(k) 5 5 id(n) ... id(k) 5 5 id(n) ???
??? ... n is k ... n is k ???
???

When one variable is assigned to another, it is the *reference value* that is assigned, not the dereferenced value.
Memory Deallocation and Garbage Collection
Next we consider what happens when in addition to variable k being reassigned, variable n is reassigned as well. The result is depicted in Figure 6-10.

FIGURE 6-10 Inaccessible Values

After n is assigned to 40, the memory location storing integer value 20 is no longer referenced— thus, it can be *deallocated*. To **deallocate** a memory location means to change its status from "currently in use" to "available for reuse." In Python, memory deallocation is automatically performed by a process

called *garbage collection*. **Garbage collection** is a method of automatically determining which locations in memory are no longer in use and deallocating them. The garbage collection process is ongoing during the execution of a Python program.

Garbage collection is a method of determining which locations in memory are no longer in use, and deallocating them.
List Assignment and Copying

Now that we understand the use of references in Python, we can revisit the discussion on copying lists from Chapter 4. We know that when a variable is assigned to another variable referencing a list, each variable ends up referring to the *same instance* of the list in memory, depicted in Figure 6-11.

FIGURE 6-11 List Assignment

Thus, any changes to the elements of list1 results in changes to list2,

... list1[0] 5 5
... list2[0]
5

We also learned that a copy of a list can be made as follows,
... list2 5 list(list1)
list() is referred to as a *list constructor*. The result of the copying is depicted in Figure 6-12.

FIGURE 6-12 Copying of Lists by Use of the List Constructor

A copy of the list structure has been made. Therefore, changes to the list elements of list1 will *not* result in changes in list2.

... list1[0] 5 5
... list2[0]
10

The situation is different if a list contains sublists, however.
... list1 5 [[10, 20], [30, 40], [50, 60]] ... list2 5 list(list1)
The resulting list structure after the assignment is depicted in Figure 6-13.

FIGURE 6-13 Shallow Copy List Structures

We see that although copies were made of the top-level list structures, the elements *within* each list were not copied. This is referred to as a **shallow copy**. Thus, if a top-level element of one list is reassigned, for example list1[0] 5 [70, 80], the other list would remain unchanged, as shown in Figure 6-14.

FIGURE 6-14 Top-Level Reassignment of Shallow Copies

If, however, a change to one of the sublists is made, for example, list1[0][0] 5 70, the corresponding change would be made in the other list. That is, list2[0][0] would be equal to 70 also, as depicted in Figure 6-15.

FIGURE 6-15 Sublevel Reassignment of Shallow Copies

A **deep copy** operation of a list (structure) makes a copy of the *complete* structure, including sublists. (Since immutable types cannot be altered, immutable parts of the structure may not be copied.) Such an operation can be performed with the deepcopy method of the copy module,

... import copy
... list2 5 copy.deepcopy(list1)
The result of this form of copying is given in Figure 6-16.

FIGURE 6-16 Deep Copy List Structures

Thus, the reassignment of any part (top level or sublist) of one list will not result in a change in the other. It is up to you as the programmer to determine which form of copy is needed for lists, and other mutable types, such as dictionaries

and sets covered in Chapter 9.

LET’S TRY IT
From the Python Shell, enter the following and observe the results.
... import copy

... list1 5 [10, 20, 30, 40] ... list2 5 list1
... id(list1) 55 id(list2) ???
... list1[0] 5 60
... list1
???
... list2
???

... list1 5 [10, 20, 30, [40]] ... list2 5 list(list1)
... id(list1) 55 id(list2) ???
... list1[0] 5 60
... list1[3][0] 5 90
... list1
???
... list2
???
... list1 5 [10, 20, 30, [40]] ... list2 5 copy.deepcopy(list1) ... id(list1) 55 id(list2)
???
... list1[0] 5 60
... list1[3][0] 5 90
... list1
???
... list2
???

... list1 5 [10, 20, 30, (40)] ... list2 5 copy.deepcopy(list1) ... list1[3][0] 5 90
???
... list1[3] 5 (100,)
... list1
???
... list2
???

The list constructor list() makes a copy of the top level of a list, in which the sublist (lower-level) structures are shared is referred to as a **shallow copy**. A **deep copy** operation makes a complete copy of a list. A deep copy operator is provided by method deepcopy of the copy module in Python.

Self-Test Questions

1. All objects have a set of _____ and _____.
2. The _____ operator is used to select members of a given object.
3. Functions that are part of an object are called _____.

4. There are two values associated with every object in Python, the _____ value and the _____ value.

5. When memory locations are *deallocated*, it means that,
(a) The memory locations are marked as unusable for the rest of the program execution. **(b)** The memory locations are marked as available for reuse during the remaining program

execution.
6. Garbage collection is the process of automatically identifying which areas of memory can be deallocated. (TRUE/FALSE)
7. Indicate which of the following is true,

(a) When one variable is assigned to another holding an integer value, if the second variable is assigned a new value, the value of the first variable will change as well.
(b) When one variable is assigned to another holding a list of integer values, if the second variable assigns a new integer value to an element in the list, the list that the first variable is assigned to will be changed as well.

ANSWERS: 1. attributes/behavior, 2. dot, 3. methods, 4. reference/dereferenced, 5. (b), 6. True, 7. (b)

6.2 Turtle Graphics

Turtle graphics refers to a means of controlling a graphical entity (a “turtle”) in a graphics window with x,y coordinates. A turtle can be told to draw lines as it travels, therefore having the ability to create various graphical designs. Turtle graphics was first developed for a language named Logo in the 1960s for teaching children how to program. Remnants of Logo still exist today.

Python provides the capability of turtle graphics in the turtle Python standard library module. There may be more than one turtle on the screen at once. Each turtle is represented by a distinct object. Thus, each can be individually controlled by the methods available for turtle objects. We introduce turtle graphics here for two reasons—first, to provide a means of better understanding objects in programming, and second, to have some fun!

Turtle graphics refers to a means of controlling a graphical entity (a “turtle”) in a graphics window with x,y coordinates.

6.2.1 Creating a Turtle Graphics Window

The first step in the use of turtle graphics is the creation of a turtle graphics window (a *turtle screen*). Figure 6-17 shows how to create a turtle screen of a certain size with an appropriate title bar. Assuming that the import turtle form of import is used, each of the turtle graphics methods must be called in the form turtle. *methodname*. The first method called, setup,

```
import turtle

# set window size
turtle.setup(800, 600)

# get reference to turtle window
window = turtle.Screen()

# set window title bar
window.title('My First Turtle Graphics Program')
```

FIGURE 6-17 Creating a Turtle Graphics Window

creates a graphics window of the specified size (in pixels). In this case, a window of size 800 pixels wide by 600 pixels high is created. The center point of the window is at coordinate (0,0). Thus, x-coordinate values to the right of the center point are positive values, and those to the left are negative values. Similarly, y-coordinate values above the center point are positive values, and those below are negative values. The top-left, top-right, bottom-left, and bottom-left coordinates for a window of size (800, 600) are as shown in Figure 6-18. A turtle graphics window in Python is also an object. Therefore, to set the title of this window, we need the reference to this object. This is done by call to method Screen.

FIGURE 6-18 Python Turtle Graphics Window (of size 800 3 600)

The background color of the turtle window can be changed from the default white background color. This is done using method bgcolor,

window 5 turtle.Screen()
window.bgcolor('blue')

See the discussion about pen color below for details on the specification of color values. The first step in the use of turtle graphics is to create a **turtle graphics window** of a specific size with an appropriate title.

6.2.2 The “Default” Turtle

A “turtle” is an entity in a turtle graphics window that can be controlled in various ways. Like the graphics window, turtles are objects. A “default” turtle is created when the setup method is called. The reference to this turtle object can be obtained by,

the_turtle 5 turtle.getturtle()

A call to getturtle returns the reference to the default turtle and causes it to appear on the screen. The initial position of all turtles is the center of the screen at coordinate (0,0), as shown in Figure 6-19.

FIGURE 6-19 The Default Turtle

The default turtle shape is an arrowhead. (The size of the turtle shape was enlarged from its default size for clarity.) A turtle's shape can be set to basic geometric shapes, or even made from a provided image file (shown in section 6.2.4).

A **default turtle** is created when the setup method is called. A call to method getturtle returns the reference to the default turtle and causes it to appear on the screen.

6.2.3 Fundamental Turtle Attributes and Behavior

Recall that objects have both attributes and behavior. Turtle objects have three fundamental attributes: *position, heading* (orientation), and *pen* attributes. We discuss each of these attributes next. Absolute Positioning

Method position returns a turtle's current position. For newly created turtles, this returns the tuple (0, 0). A turtle's position can be changed using *absolute positioning* by moving the turtle to a specific x,y coordinate location by use of method setposition. An example of this is given in Figure 6-20.

```
# set window title
window = turtle.Screen()
window.title('Absolute Positioning')

# get default turtle and hide
the_turtle = turtle.getturtle()
the_turtle.hideturtle()

# create square (absolute positioning)
the_turtle.setposition(100, 0)
the_turtle.setposition(100, 100)
the_turtle.setposition(0, 100)
the_turtle.setposition(0, 0)

# exit on close window
turtle.exitonclick()
```

FIGURE 6-20 Absolute Positioning of Turtle

The turtle is made invisible by a call to method hideturtle. Since newly created turtles are positioned at coordinates (0, 0), the square will be displayed near the middle of the turtle window. To draw the square, the turtle is first positioned at coordinates (100, 0), 100 pixels to the right of its current position. Since the turtle's pen is down, a line will be drawn from location (0, 0) to location (100, 0). The turtle is then positioned at coordinates (100, 100), which draws a line from the bottom-right corner to the top-right corner of the square. Positioning the turtle to coordinates (0, 100) draws a line from the top-right corner to the top-left corner. Finally, positioning the turtle back to coordinates (0, 0) draws the final line from the top-left corner to the bottom-left corner.

A turtle's position can be changed using *absolute positioning* by use of method setposition.

Turtle Heading and Relative Positioning

A turtle's position can also be changed through *relative positioning*. In this case, the location that a turtle moves to is determined by its second fundamental attribute, its heading. A newly created turtle's heading is to the right, at 0 degrees. A turtle with heading 90 degrees moves up; with a heading 180 degrees moves left; and with a heading 270 degrees moves down. A turtle's heading can be changed by turning the turtle a given number of degrees left, left(90), or right,

right(90). The forward method moves a turtle in the direction that it is currently heading. An example of relative positioning is given in Figure 6-21.

```
# set window title
window = turtle.Screen()
window.title('Relative Positioning')

# get default turtle and hide
the_turtle = turtle.getturtle()
the_turtle.hideturtle()

# create box (relative positioning)
the_turtle.forward(100)
the_turtle.left(90)
the_turtle.forward(100)
the_turtle.left(90)
the_turtle.forward(100)
the_turtle.left(90)
the_turtle.forward(100)

# exit on close window
turtle.exitonclick()
```

FIGURE 6-21 Relative Positioning of Turtle

In this example, the turtle is controlled using relative positioning, drawing the same square as in Figure 6-20 above. Since turtles are initially positioned at coordinates (0, 0) with an initial heading of 0 degrees, the first step is to move the turtle forward 100 pixels. That draws the bottom line of the square. The turtle is then turned left 90 degrees and again moved forward 100 pixels. This draws the line of the right side of the square. These steps continue until the turtle arrives back at the original coordinates (0, 0), completing the square.

Methods left and right change a turtle's heading relative to its current heading. A turtle's heading can also be set to a specific heading by use of method setheading: the_turtle.setheading(90). In addition, method heading can be used to determine a turtle's current heading.

A turtle's position can be changed using *relative positioning* by use of methods setheading, left, right, and forward.

Pen Attributes

The pen attribute of a turtle object is related to its drawing capabilities. The most fundamental of these attributes is whether the pen is currently "up" or "down," controlled by methods penup() and pendown(). When the pen attribute value is "up," the turtle can be moved to another location without lines being drawn. This

is especially needed when drawing graphical images with disconnected segments. Example use of these methods is given in Figure 6-22.

In this example, the turtle is hidden so that only the needed lines appear. Since the initial location of the turtle is at coordinate (0, 0), the pen is set to “up” so that the position of the turtle can be set to (2100, 0) without a line being drawn as it moves. This puts the turtle at the bottom of the left side of the letter. The pen is then set to “down” and the turtle is moved to coordinate (0, 250), drawing as it moves. This therefore draws a line from the bottom of the left side to the top of the “A.” The turtle is then moved (with its pen still down) to the location of the bottom of the right side of the letter, coordinate (100, 0). To cross the “A,” the pen is again set to “up” and the turtle is moved to the location of the left end of the crossing line, coordinate (264, 90). The pen is then set to “down” and moved to the end of the crossing line, at coordinate (64, 90), to finish the letter.

```
turtle.setup(800, 600)
window = turtle.Screen()
window.title('Drawing an A')

the_turtle = turtle.getturtle()
the_turtle.hideturtle()
the_turtle.penup()
the_turtle.setposition(-100, 0)
the_turtle.pendown()
the_turtle.setposition(0, 250)
the_turtle.setposition(100, 0)
the_turtle.penup()
the_turtle.setposition(-64, 90)
the_turtle.pendown()
the_turtle.setposition(64, 90)

turtle.exitonclick()
```

FIGURE 6-22 Example Use of Methods penup and pendown

The pen size of a turtle determines the width of the lines drawn when the pen attribute is “down.” The pensize method is used to control this: the_turtle.pensize(5). The width is given in pixels, and is limited only by the size of the turtle screen. Example pen sizes are depicted in Figure 6-23.

FIGURE 6-23 Example Turtle Pen Sizes

The pen color can also be selected by use of the pencolor method: the_turtle.pencolor('blue'). The name of any common color can be used, for example 'white', 'red', 'blue', 'green', 'yellow', 'gray', and 'black'. Colors can also be specified in RGB (red/green/blue) component values. These values can be specified in the range 0–255 if the color mode attribute of the turtle window is set as given below,

```
turtle.colormode(255)
the_turtle.pencolor(238, 130, 238) # violet
```

This provides a means for a full spectrum of colors to be displayed.
The pen attributes that can be controlled include whether the pen is down or up (using methods penup and pendown), the pen size (using method pensize), and the pen color (using method pencolor).

6.2.4 Additional Turtle Attributes

In addition to the fundamental turtle attributes already discussed, we provide details on other attributes of a turtle that may be controlled. This includes whether the turtle is visible or not, the size (both demonstrated above), shape, and fill color of the turtle, the turtle's speed, and the tilt of the turtle. We will discuss each of these attributes next.

Turtle Visibility

As we saw, a turtle's visibility can be controlled by use of methods hideturtle() and showturtle() (in which an invisible turtle can still draw on the screen). There are various reasons for doing this. A turtle may be made invisible while being repositioned on the screen. In gaming, a turtle might be made invisible when it meets its "demise." Or maybe a given turtle needs to blink, as we will see at the end of the chapter.

Methods showturtle() and hideturtle()control a turtle's visibility.

Turtle Size

The size of a turtle shape can be controlled with methods resizemode and turtlesize as shown in Figure 6-24.

```
# set to allow user to change turtle size
the_turtle.resizemode('user')

# set a new turtle size
the_turtle.turtlesize(3, 3)
```

FIGURE 6-24 Changing the Size of a Turtle

The first instruction sets the resize attribute of a turtle to 'user'. This allows the user (programmer) to change the size of the turtle by use of method turtlesize. Otherwise, calls to turtlesize will have no effect. The call to method turtlesize in the figure is passed two parameters. The first is used to change the *width* of the shape (perpendicular to its orientation), and the second changes its *length* (parallel to its orientation). Each value provides a factor by which the size is to be changed. Thus, the_turtle.turtlesize(3, 3)stretches both the width and length of the current turtle shape by a factor of 3. (A third parameter can also be added that determines the thickness of the shape's outline.)

There are two other values that method resizemode may be set to. An argument value of 'auto' causes the size of the turtle to change with changes in the pen size, whereas a value of 'noresize' causes the turtle shape to remain the same size.

The size of a given turtle shape can be controlled with methods resizemode and turtlesize.

Turtle Shape

There are a number of ways that a turtle's shape (and fill color) may be defined to something other than the default shape (the arrowhead) and fill color (black). First, a turtle may be assigned one of the following provided shapes: 'arrow', 'turtle', 'circle', 'square', 'triangle', and 'classic' (the default arrowhead shape), as

shown in Figure 6-25.

FIGURE 6-25 Available Turtle Shapes

The shape and fill colors are set by use of the shape and fillcolor methods, the_turtle.shape('circle') the_turtle.fillcolor('white')

New shapes may be created and registered with (added to) the turtle screen's *shape dictionary*. One way of creating a new is shape by providing a set of coordinates denoting a polygon, as shown in Figure 6-26.

```
turtle.setup(800, 600)
window = turtle.Screen()
window.title('My Polygon')
the_turtle = turtle.getturtle()

turtle.register_shape('mypolygon',
((0, 0), (100, 0), (140, 40)))

the_turtle.shape('mypolygon')
the_turtle.fillcolor('white')
```

FIGURE 6-26 Creating a New Polygon Turtle Shape

In the figure, method register_shape is used to register the new turtle shape with the name mypolygon. The new shape is provided by the tuple of coordinates in the second argument. These coordinates define the polygon shown in the figure. Once the new shape is defined, a turtle can be set to that shape by calling the shape method with the desired shape's name. The fillcolor method is then called to make the fill color of the polygon white (with the edges remaining black). It is also possible to create turtle shapes composed of various individual polygons called *compound shapes*. We refer the reader to the official online Python documentation of the turtle module for details (see http://docs.python.org/py3k/library/turtle. html#module-turtle).

The creation of this polygon may not seem too exciting, but the orientation of a turtle can be changed. In addition, a turtle is able to *stamp* its shape on the screen, which remains there even after the turtle is repositioned (or relocated).

That means that we can create all sorts interesting graphic patterns by appropriately repositioning the turtle, as shown in Figure 6-27.

```
turtle.setup(800, 600)
window = turtle.Screen()
window.title('My Polygon Design')
the_turtle = turtle.getturtle()

turtle.register_shape('mypolygon',
((0, 0), (100, 0), (140, 40)))
the_turtle.shape('mypolygon')
the_turtle.fillcolor('white')

for angle in range(0, 360, 10):
    the_turtle.setheading(angle)
    the_turtle.stamp()
```

FIGURE 6-27 Creating a Design from a Turtle using a Polygon Shape

Only a few lines of code are needed to generate this design. The for loop in the figure iterates variable angle over the complete range of degrees, 0 to 360, by increments of 10 degrees. Within the loop the turtle's heading is set to the current angle, and the stamp() method is called to stamp the polygon shape at the turtle's current position. By varying the shape of the polygon and the angles that the turtle is set to, a wide range of such designs may be produced.

Another way that a turtle shape can be created is by use of an image. The image file used must be a "gif file" (with file extension .gif). The name of the file is then registered and the shape of the turtle set to the registered name,

register_shape('image1.gif') the_turtle.shape('image1.gif')

The final program of this chapter gives an example of the use of image shapes. A turtle's shape may be set to one of the provided shapes, a described polygon (or collection of polygons), or an image.

Turtle Speed

At times, you may want to control the speed at which a turtle moves. A turtle's speed can be set to a range of speed values from 0 to 10, with a "normal" speed being around 6. To set the speed of the turtle, the speed method is used, the_turtle.speed(6). The following speed values can be set using a descriptive

rather than a numeric value,

10: 'fast' 6: 'normal' 3: 'slow' 1: 'slowest' 0: 'fastest'

Thus, a normal speed can also be set by the_turtle.speed('normal'). When using the turtle for line drawing only, the turtle will move more quickly if it is made invisible (by use of the hideturtle method).

The speed of a turtle can be controlled by use of the speed method.

6.2.5 Creating Multiple Turtles

So far, we have seen examples in which there is only one turtle object, the default turtle created with a turtle window. However, it is possible to create and control any number of turtle objects. To create a new turtle, the Turtle() method is used,

turtle1 5 turtle.Turtle() turtle2 5 turtle.Turtle() etc.

By storing turtle objects in a list, any number of turtles may be maintained,

turtles 5 []
turtles.append(turtle.Turtle()) turtles.append(turtle.Turtle()) etc.

An example of using multiple turtle objects is given in the following “Let’s Apply It” section.
Any number of turtle objects can be created by use of method Turtle().

6.2.6 Let’s Apply It—Bouncing Balls Program

Following is a program (Figure 6-29) that displays one or more bouncing balls within a turtle screen. This program utilizes the following programming features.
➤ turtle module ➤ time module ➤ random module
Example execution of the program is given in Figure 6-28.

```
Program Execution ...

This program simulates one or more bouncing balls on a turtle screen.
Enter number of seconds to run: 30
Enter number of balls in simulation: 5
```

FIGURE 6-28 Execution of Bouncing Balls Program

In addition to the turtle graphics module, this program makes use of the time and random Python standard library modules to allow control of how long (in seconds) the simulation is executed, as indicated by the user, and to generate the random motion of the bouncing balls.

The main section of the program begins on **line 52** with the programming greeting. On **lines 58–60**, the size of the turtle screen (in pixels) is hard-coded into the program, assigned to variables screen_width and screen_height. Since all references to the screen size are through these variables, the desired window size can be altered by simply altering these variables.

```
# Bouncing Balls Simulation Program

import turtle
import random
import time

def atLeftEdge(ball, screen_width):
    if ball.xcor() < -screen_width / 2:
        return True
    else:
        return False

def atRightEdge(ball, screen_width):
    if ball.xcor() > screen_width / 2:
        return True
    else:
        return False

def atTopEdge(ball, screen_height):
    if ball.ycor() > screen_height / 2:
        return True
    else:
        return False

def atBottomEdge(ball, screen_height):
    if ball.ycor() < -screen_height / 2:
        return True
    else:
        return False

def bounceBall(ball, new_direction):
    if new_direction == 'left' or new_direction == 'right':
        new_heading = 180 - ball.heading()
    elif new_direction == 'down' or new_direction == 'up':
        new_heading =  360 - ball.heading()

    return new_heading

def createBalls(num_balls):
    balls = []
    for k in range(0, num_balls):
        new_ball = turtle.Turtle()
        new_ball.shape('circle')
        new_ball.fillcolor('black')
        new_ball.speed(0)
        new_ball.penup()
        new_ball.setheading(random.randint(1, 359))
        balls.append(new_ball)

    return balls

```

FIGURE 6-29 Bouncing Balls Program (*Continued*)

On **lines 63–64**, the turtle screen is created and its reference value assigned to variable window. The title of the window is assigned through a call to the title method. Following that, the user is prompted to enter the number of seconds for

the simulation, as well as the number of simultaneously bouncing balls.

```
52 # ---- main
53 # program greeting
54 print('This program simulates bouncing balls in a turtle screen')
55 print('for a specified number of seconds')
56
57 # init screen size
58 screen_width = 800
59 screen_height = 600
60 turtle.setup(screen_width, screen_height)
61
62 # create turtle window
63 window = turtle.Screen()
64 window.title('Bouncing Balls')
65
66 # prompt user for execution time and number of balls
67 num_seconds = int(input('Enter number of seconds to run: '))
68 num_balls = int(input('Enter number of balls in simulation: '))
69
70 # create balls
71 balls = createBalls(num_balls)
72
73 # set start time
74 start_time = time.time()
75
76 # begin simulation
77 terminate = False
78
79 while not terminate:
80     for k in range(0, len(balls)):
81         balls[k].forward(15)
82
83         if atLeftEdge(balls[k], screen_width):
84             balls[k].setheading(bounceBall(balls[k], 'right'))
85         elif atRightEdge(balls[k], screen_width):
86             balls[k].setheading(bounceBall(balls[k], 'left'))
87         elif atTopEdge(balls[k], screen_height):
88             balls[k].setheading(bounceBall(balls[k], 'down'))
89         elif atBottomEdge(balls[k], screen_height):
90             balls[k].setheading(bounceBall(balls[k], 'up'))
91
92         if time.time() - start_time > num_seconds:
93             terminate = True
94
95 # exit on close window
96 turtle.exitonclick()
```

FIGURE 6-29 Bouncing Balls Program

Function createBalls is called (on **line 71**) to create and return a list of turtle objects with a ball shape. The function definition (**lines 39–50**) initializes an empty list named balls and creates the requested number of balls one-by-one, each appended to the list, by use of the for loop at **line 41**. Each ball is created with shape 'circle', fill color of 'black', speed of 0 (fastest speed), and with pen

attribute 'up'. In addition, the initial heading of each turtle is set to a random angle between 1 and 359 (**line 47**).

Back in the main program section at **line 74**, the current time (in seconds) is obtained from a call to method time of the time module: time.time(). The current time value is stored in variable start_time. (The current time is the number of seconds since the “epoch,” which is January 1, 1970. This will be discussed further in the Horse Racing program that follows.) The while loop beginning on **line 79** begins the simulation. The loop iterates as long as Boolean variable terminate is False (initialized to False on **line 77**). The for loop at **line 80** moves each of the specified number of balls a small distance until reaching one of the four edges of the window (left, right, top, or bottom edge). Boolean functions atLeftEdge, atRightEdge, atTopEdge, and atBottomEdge are used to determine when a ball is at an edge (defined in **lines 7–29**). Function bounceBall is called to bounce the ball in the opposite direction it is heading, and returns the new heading of the ball, passed as the argument to that ball’s setheading method. Finally, on **line 92** a check is made to determine whether the user-requested simulation time has been exceeded. If so, Boolean variable terminate is set to True, and the program terminates. Because of the call to exitonclick() on **line 96**, the program will properly shut down when the close button of the turtle window is clicked.

Self-Test Questions **1.** A turtle screen is an 800-pixel wide by 600-pixel high graphics window. (TRUE/FALSE).
2. The three main attributes of a turtle object are _____, _____, and _____.

3. A turtle can be moved using either _____ or _____ positioning.
4. A turtle can only draw lines when it is not hidden. (TRUE/FALSE)

5. A turtle shape is limited to an arrow, turtle, circle, square, triangle, or classic (default) shape. (TRUE/FALSE)

6. What attribute of a turtle determines the size of the lines it draw?
(a) Pen size
(b) Turtle size

7. A turtle can draw in one of seven colors. (TRUE/FALSE)

8. A turtle can leave an imprint of its shape on the screen by use of the _____ method.

9. In order to create a new turtle object, the _____ method is called.

ANSWERS: 1. False, 2. position, heading, pen, 3. absolute/relative 4. False, 5. False, 6. (a), 7. False, 8. stamp, 9. Turtle

COMPUTATIONAL PROBLEM SOLVING 6.3 Horse Race Simulation Program

In this section, we design, implement and test a program that simulates a horse race.

6.3.1 The Problem

The problem is to create a visualization of a horse race in which horses are moved ahead a random distance at fixed intervals until there is a winner, as shown in Figure 6-30.

FIGURE 6-30 Example Horse Race Simulation

6.3.2 Problem Analysis

The program needs a source of random numbers for advancing the horses a random distance in the race. We can use the random number generator of the Python standard library module random that we used in Chapter 3 in the Coin Change Exercise example. The remaining part of the problem is in the creation of appropriate graphics for producing a visualization of a horse race. We shall make use of the turtle graphics module from the Python standard library to do

this.

6.3.3 Program Design

Meeting the Program Requirements

There are no specific requirements for this problem, other than to create an appropriate simulation of a horse race. Therefore, the requirement is essentially the generation of a horse race in which the graphics look sufficiently compelling, and each horse has an equal chance of winning a given race. Since a specific number of horses was not specified, we will design the program for ten horses in each race.

Data Description

The essential information for this program is the current location of each of the ten horses in a given race. Each turtle is an object, whose attributes include its shape and its coordinate position on the turtle screen. Therefore, we will maintain a list of ten turtle objects with the shape attribute of a horse image for this purpose. Thus, suitable horse images must be found or created for this purpose.

Algorithmic Approach

There is no algorithm, per se, needed in this program other than to advance each horse a random distance at fixed time intervals until one of the horses reaches a certain point on the turtle screen (the “finish line”).

Overall Program Steps

The overall steps in this program design are given in Figure 6-31.

FIGURE 6-31 Overall Steps of the Horse Race Simulation Program

6.3.4 Program Implementation and Testing
Stage 1—Creating an Initial Turtle Screen Layout

We first develop and test an initial program that lays out the positions of the starting horses on the turtle graphics screen, as shown in Figure 6-32. Figure 6-33 provides this first stage of the program.

FIGURE 6-32
Output of Stage 1 of the Horse Race Simulation Program

At **line 3** the turtle module is imported. Since the import *module_name* form of import is used, each call to a method of this module must be prefixed with the module name. For example, turtle. setup(750, 800) on **line 31** (which sets the turtle screen size to a width of 750 and a height of 800 pixels).

The intent of this version of the program is to ensure that the turtle screen is appropriately sized and that the initial layout of horse locations is achieved. Therefore, only the default turtle shape is used at this point. In the next version we will focus on generating a set of horse images on the screen. Thus, on **line 34**, the turtle screen object is retrieved (by the call to turtle. Sreen()) and its reference assigned to variable window. The start location of the first (lowest)

horse is set to an x coordinate value of 240, and a y coordinate value of 2200. This puts the turtle screen object at the lower right corner of the screen. The amount of vertical separation between the horses is assigned to variable track_separation.These values were determined from knowledge of the screen coordinates in turtle graphics and a little trial and error.

Next, on **line 44** a call is made to function generateHorses (at **lines 9–15**). This function returns a list of ten new turtle objects, and assigned to variable horses. Function newHorse (**lines 5–7**) is called by function generateHorses to create each new horse turtle object. At this stage, function newHorse simply creates and returns a regular turtle object. In the next stage however, it will be responsible for returning new turtle objects with an appropriate horse shape.

The position for each of these horses is determined by function placeHorses on **lines 17– 23**. It is passed the list of horse turtle objects, the location of the first turtle, and the amount of separation between each (established as 60 pixels on **line 41**). Function placeHorses, therefore,

```python
# Horse Racing Program (Stage 1)

import turtle

def newHorse():
    horse = turtle.Turtle()
    return horse

def generateHorses(num_horses):
    horses = []
    for k in range(0, num_horses):
        horse = newHorse()
        horses.append(horse)

    return horses

def placeHorses(horses, loc, separation):
    for k in range(0, len(horses)):
        horses[k].hideturtle()
        horses[k].penup()
        horses[k].setposition(loc[0], loc[1] + k * separation)
        horses[k].setheading(180)
        horses[k].showturtle()

# ---- main

# init number of horses
num_horses = 10

# set window size
turtle.setup(750, 800)

# get turtle window
window = turtle.Screen()

# set window title bar
window.title('Horse Race Simulation Program')

# init screen layout parameters
start_loc = (240, -200)
track_separation = 60

# generate and init horses
horses = generateHorses(num_horses)

# place horses at starting line
placeHorses(horses, start_loc, track_separation)

# terminate program when close window
turtle.exitonclick()
```

FIGURE 6-33 Stage 1 of the Horse Race Simulation Program

contains a for loop that iterates over the list of horse objects and makes them

initially hidden with their pen up (**lines 19–20**), moves each to its starting position (**line 21**), sets the heading of each to 180 degrees to move left (**line 22**), and then makes each visible (**line 23**). Finally, method exitonclick() is called so that the program will terminate when the user clicks on the program window's close box.

In the next stage, we further develop the program to include the specific shapes and images for the simulation.

Stage 2—Adding the Appropriate Shapes and Images

We next develop and test the program with additional code that adds the horse shapes (images) needed. The resulting turtle screen is shown in Figure 6-34. Figure 6-35 shows this second stage of the program.

FIGURE 6-34

Output of Stage 2 of the Horse Race Simulation Program

In this stage of the program we add functions getHorseImages and registerHorseImages, called from **lines 61** and **62** of the main program section. Function getHorseImages returns a list of GIF image files. Each image contains the same horse image, each with a unique number 1 to 10. Function registerHorseImages does the required registering of images in turtle graphics by calling method turtle.register_shape on each.

Function generateHorses (**lines 26–32**) is implemented the same way as in stage 1 to return a list of horse turtle objects, except that it is altered to be passed an argument containing a list of horse images. Thus, the call to generateHorses in **line 65** is altered to pass the list of images in variable horse_images. Function newHorse (**lines 19–24**) is altered as well to be passed a particular horse image for the horse that is created, horse.shape(image_file).

Stage 3—Animating the Horses

Next we develop and test the program with additional code that animates the horses so that they are randomly advanced until a horse crosses the finish line. The resulting turtle screen is shown in Figure 6-36. Figure 6-37 provides this third stage of the program.

```
# Horse Racing Program (Stage 2)

import turtle

def getHorseImages(num_horses):
    # init empty list
    images = []

    # get all horse images
    for k in range(0, num_horses):
        images = images + ['horse_' + str(k + 1) + '_image.gif']

    return images

def registerHorseImages(images):
    for k in range(0, len(images)):
        turtle.register_shape(images[k])

def newHorse(image_file):
    horse = turtle.Turtle()
    horse.hideturtle()
    horse.shape(image_file)

    return horse

def generateHorses(images, num_horses):
    horses = []
    for k in range(0, num_horses):
        horse = newHorse(images[k])
        horses.append(horse)

    return horses

def placeHorses(horses, loc, separation):
    for k in range(0, len(horses)):
        horses[k].hideturtle()
        horses[k].penup()
        horses[k].setposition(loc[0], loc[1] + k * separation)
        horses[k].setheading(180)
        horses[k].showturtle()
```

FIGURE 6-35 Stage 2 of the Horse Simulation Race Program (*Continued*)

Two new functions are added in this version of the program, startHorses and displayWinner. Function startHorses (**lines 44–58**) is passed the list of horse turtle objects, the location of the finish line (as an x coordinate value on the turtle screen) and the fundamental increment amount—each horse is advanced by one to three times this amount. The while loop for incrementally moving the horses is on **line 49**. The loop iterates until a winner is found, that is, until the

variable have_winner is True. Therefore, have_winner is initialized to False in **line 46**. Variable k, initialized on **line 48**, is used to index into the list of horse turtle objects. Since each horse in turn is advanced some amount during the race, variable k is incremented by one, modulo the number of horses in variable num_horses (10) (**line 57**). When k becomes equal to num_horses 21 (9), it is reset to 0 (for horse 1).

The amount that each horse is advanced is a factor of one to three randomly determined by call to method randint(1,3)of the Python standard library module random in **line 51**. Variable forward_incr is multiplied by this factor to move the horses forward an appropriate amount.

```
42 # ---- main
43
44 # init number of horses
45 num_horses = 10
46
47 # set window size
48 turtle.setup(750, 800)
49
50 # get turtle window
51 window = turtle.Screen()
52
53 # set window title bar
54 window.title('Horse Race Simulation Program')
55
56 # init screen layout parameters
57 start_loc = (240, -200)
58 track_separation = 60
59
60 # register images
61 horse_images = getHorseImages()
62 registerHorseImages(horse_images)
63
64 # generate and init horses
65 horses = generateHorses(horse_images)
66
67 # place horses at starting line
68 placeHorses(horses, start_loc, track_separation)
69
70 # terminate program when close window
71 turtle.exitonclick()
```

FIGURE 6-35 Stage 2 of the Horse Simulation Race Program

FIGURE 6-36 Output of Stage 3 of the Horse Race Simulation Program

```python
# Horse Racing Program (Stage 3)

import turtle
import random

def getHorseImages(num_horses):
    # init empty list
    images = []

    # get all horse images
    for k in range(0, num_horses):
        images = images + ['horse_' + str(k+1) + '_image.gif']

    return images

def registerHorseImages(images):
    for k in range(0, len(images)):
        turtle.register_shape(images[k])

def newHorse(image_file):
    horse = turtle.Turtle()
    horse.hideturtle()
    horse.shape(image_file)

    return horse

def generateHorses(images, num_horses):
    horses = []
    for k in range(0, num_horses):
        horse = newHorse(images[k])
        horses.append(horse)

    return horses

def placeHorses(horses, loc, separation):
    for k in range(0, len(horses)):
        horses[k].hideturtle()
        horses[k].penup()
        horses[k].setposition(loc[0], loc[1] + k * separation)
        horses[k].setheading(180)
        horses[k].showturtle()
        horses[h].pendown()

def startHorses(horses, finish_line, forward_incr):
    # init
    have_winner = False

    k = 0
    while not have_winner:
        horse = horses[k]
        horse.forward(random.randint(1, 3) * forward_incr)

        # check for horse over finish line
        if horse.position()[0] < finish_line:
            have_winner = True
        else:
            k = (k + 1) % len(horses)
    return k
```

FIGURE 6-37 Stage 3 of the Horse Race Simulation Program (*Continued*)

```
60  def displayWinner(winning_horse):
61      print('Horse', winning_horse, 'the winner!')
62
63  # ---- main
64
65  # init number of horses
66  num_horses = 10
67
68  # set window size
69  turtle.setup(750, 800)
70
71  # get turtle window
72  window = turtle.Screen()
73
74  # set window title bar
75  window.title('Horse Race Simulation Program')
76
77  # init screen layout parameters
78  start_loc = (240, -200)
79  finish_line = -240
80  track_separation = 60
81  forward_incr = 6
82
83  # register images
84  horse_images = getHorseImages(num_horses)
85  registerHorseImages(horse_images)
86
87  # generate and init horses
88  horses = generateHorses(horse_images, num_horses)
89
90  # place horses at starting line
91  placeHorses(horses, start_loc, track_separation)
92
93  # start horses
94  winner = startHorses(horses, finish_line, forward_incr)
95
96  # display winning horse
97  displayWinner(winner + 1)
98
99  # terminate program when close window
100 turtle.exitonclick()
```

FIGURE 6-37 Stage 3 of the Horse Race Simulation Program

The value of forward_incr is initialized in the main program section. This value can be adjusted to speed up or slow down the overall speed of the horses.

Function displayWinner displays the winning horse number in the Python shell (**lines 60–61**). This function will be rewritten in the next stage of program

development to display a “winner” banner image in the turtle screen. Thus, this implementation of the function is for testing purposes only.

The main program section (**lines 63–100**) is the same as in the previous stage of program development, except for the inclusion of the calls to functions startHorses and displayWinner in **lines 94** and **97**.

Final Stage—Adding Race Banners

Finally, we add the code for the displaying of banners at various points in the race as shown earlier in Figure 6-30. In Figure 6-38 is the final stage of the program. This final version imports one additional module, Python Standard Library module time (**line 5**), used to control the blink rate of the winning horse.

While the race progresses within the while loop at **line 102**, checks for the location of the lead horse are made in two places—before and after the halfway mark of the race (on **line 108**). If the

```
# Horse Racing Program (Final Stage)

import turtle
import random
import time

def getHorseImages(num_horses):
    # init empty list
    images = []

    # get all horse images
    for k in range(0, num_horses):
        images = images + ['horse_' + str(k + 1) + '_image.gif']

    return images

def getBannerImages(num_horses):
    # init empty list
    all_images = []

    # get "They're Off" banner image
    images = ['theyre_off_banner.gif']
    all_images.append(images)

    # get early lead banner images
    images = []
    for k in range(0, num_horses):
        images = images + ['lead_at_start_' + str(k + 1) + '.gif']
    all_images.append(images)

    # get mid-way lead banner images
    images = []
    for k in range(0, num_horses):
        images = images + ['looking_good_' + str(k + 1) + '.gif']
    all_images.append(images)

    # get "We Have a Winner" banner image
    images = ['winner_banner.gif']
    all_images.append(images)

    return all_images

def registerHorseImages(images):
    for k in range(0, len(images)):
        turtle.register_shape(images[k])

```

FIGURE 6-38 Final Stage of the Horse Race Simulation Program (*Continued*)

```
47 def registerBannerImages(images):
48     for k in range(0, len(images)):
49         for j in range(0, len(images[k])):
50             turtle.register_shape(images[k][j])
51
52 def newHorse(image_file):
53     horse = turtle.Turtle()
54     horse.hideturtle()
55     horse.shape(image_file)
56
57     return horse
58
59 def generateHorses(images, num_horses):
60     horses = []
61     for k in range(0, num_horses):
62         horse = newHorse(images[k])
63         horses.append(horse)
64
65     return horses
66
67 def placeHorses(horses, loc, separation):
68     for k in range(0, len(horses)):
69         horses[k].hideturtle()
70         horses[k].penup()
71         horses[k].setposition(loc[0], loc[1] + k * separation)
72         horses[k].setheading(180)
73         horses[k].showturtle()
74         horses[k].pendown()
75
76 def findLeadHorse(horses):
77     # init
78     lead_horse = 0
79
80     for k in range(1, len(horses)):
81         if horses[k].position()[0] <  \
82            horses[lead_horse].position()[0]:
83             lead_horse = k
84     return lead_horse
85
86 def displayBanner(banner, position):
87     the_turtle = turtle.getturtle()
88     the_turtle.setposition(position[0], position[1])
89     the_turtle.shape(banner)
90     the_turtle.stamp()
91
```

FIGURE 6-38 Final Stage of the Horse Race Simulation Program (*Continued*)

x coordinate location of the lead horse is less then 125, the “early lead banner” is displayed on **line 117** by a call to function displayBanner. Otherwise, if one second has elapsed, then the “midrace lead banner” is displayed on **line 111**.

The sleep method of the time module is used to control the blinking of the winning horse in function displayWinner. A “count-down” variable, blink_counter, is set to 5 on **line 133**. This will cause the winning horse to blink five times. The following while loop decrements

```
92 def startHorses(horses, banners, finish_line, forward_incr):
93     # init
94     have_winner = False
95     early_leading_horse_displayed = False
96     midrace_leading_horse_displayed = False
97
98     # display "They're Off" banner image
99     displayBanner(banner_images[0][0], (70, -300))
100
101     k = 0
102     while not have_winner:
103         horse = horses[k]
104         horse.forward(random.randint(1, 3) * forward_incr)
105
106         # display mid-race lead banner
107         lead_horse = findLeadHorse(horses)
108         if horses[lead_horse].position()[0] < -125 and \
109            not midrace_leading_horse_displayed:
110
111             displayBanner(banners[2][lead_horse], (40, -300))
112             midrace_leading_horse_displayed = True
113
114         # display early lead banner
115         elif horses[lead_horse].position()[0] < 125 and \
116            not early_leading_horse_displayed:
117             displayBanner(banners[1][lead_horse], (10, -300))
118             early_leading_horse_displayed = True
119
120         # check for horse over finish line
121         if horse.position()[0] < finish_line:
122             have_winner = True
123         else:
124             k = (k + 1) % len(horses)
125     return k
126
```

FIGURE 6-38 Final Stage of the Horse Race Simulation Program (*Continued*)

blink_counter and continues to iterate until blink_counter is 0. Variable show, initialized to False on **line 132**, is used to alternately show and hide the turtle based on its current (Boolean) value, which is toggled back and forth between True and False each time through the loop. The sleep method is called on **line 143** to cause the program execution to suspend for four-tenths of a second so that the switch between the visible and invisible horse appears slowly enough to

cause a blinking effect. This version of displayWinner replaces the previous version that simply displayed the winning horse number in the Python shell window.

Added functions getBannerImages (**lines 17–41**), registerBannerImages (**lines 47–50**), and displayBanner (**lines 86–90**) incorporate the banner images into the program the same way that the horse images were incorporated in the previous program version. Function startHorses was modified to take another parameter, banners, containing the list of registered banners displayed during the race, passed to it from the main program section.

```
127 def displayWinner(winning_horse, winner_banner):
128     # display "We Have a Winner" banner
129     displayBanner(winner_banner, (20, -300))
130 
131     # blink winning horse
132     show = False
133     blink_counter = 5
134     while blink_counter != 0:
135         if show:
136             winning_horse.showturtle()
137             show = False
138             blink_counter = blink_counter - 1
139         else:
140             winning_horse.hideturtle()
141             show = True
142 
143         time.sleep(.4)
144 
145 # ---- main
146 
147 # init number of horses
148 num_horses = 10
149 
150 # set window size
151 turtle.setup(750, 800)
152 
153 # get turtle window
154 window = turtle.Screen()
155 
156 # set window title
157 window.title('Horse Race Simulation Program')
158 
159 # hide default turtle and keep from drawing
160 the_turtle.hideturtle()
161 the_turtle.penup()
162 
163 # init screen layout parameters
164 start_loc = (240, -200)
165 finish_line = -240
166 track_separation = 60
167 forward_incr = 6
168 
169 # register images
170 horse_images = getHorseImages()
171 banner_images = getBannerImages()
172 registerHorseImages(horse_images)
173 registerBannerImages(banner_images)
174 
```

FIGURE 6-38 Final Stage of the Horse Race Simulation Program (*Continued*)

Finally, the default turtle (created with the turtle graphics window) is utilized in

function displayBanners and in the main section. It is used to display the various banners at the bottom of the screen. To do this, the turtle's "shape" is changed to the appropriate banner images stored in list banner_images. To prevent the turtle from drawing lines when moving from the initial (0, 0)coordinate location to where banners are displayed, the default turtle is hidden and its pen attribute is set to "up" (**lines 160–161**).


```
175 # generate and init horses
176 horses = generateHorses(horse_images)
177
178 # place horses at starting line
179 placeHorses(horses, start_loc, track_separation)
180
181 # start horses
182 winner = startHorses(horses, banner_images, finish_line,
183                      forward_incr)
184
185 # light up for winning horse
186 displayWinner(horses[winner], banner_images[3][0])
187
188 # terminate program when close window
189 turtle.exitonclick()
```

FIGURE 6-38 Final Stage of the Horse Race Simulation Program

CHAPTER SUMMARY General Topics

Software Objects/Methods
References/Reference vs. Dereferenced Values Reference Assignment
Memory Allocation/Deallocation
Garbage Collection
Shallow vs. Deep Copy Operations

Python-Specific Programming Topics Objects and Turtle Graphics in Python
CHAPTER EXERCISES Section 6.1
1. Indicate exactly what the contents of lst1 and lst2 would be after each of the following set of assignments,

(a) lst1 5 [10, 20, 30] **(b)** lst1 5 [10, 20, 30] **(c)** lst1 5 [10, 20, 30] lst2 5 [10, 20, 30] lst2 5 lst1 lst2 5 list(lst1) lst1[2] 5 50 lst1[2] lst1[2]

2. Indicate which of the following set of assignments would result in automatic

garbage collection in Python.

(a) lst1 5 [1, 2, 3] **(b)** str1 5 'Hello World' **(c)** tuple1 5 (1, 2, 3) lst2 5 [5, 6, 7] str2 ' Nice Day ' tuple2 tuple1 lst1 lst2 str3 tuple1

3. For the set of assignments in question 1, indicate how both the id method and is operator can be used to determine if lists lst1 and lst2 are each referencing the same list instance in memory.
Section 6.2
4. Give a set of instructions to create a turtle window of size 400 pixels wide and 600 pixels high, with a title of 'Turtle Graphics Window'.

5. Give a set of instructions that gets the default turtle and sets it to an actual turtle shape. **6.** For each of the following method calls on turtle the_turtle, indicate in what part of the screen the turtle will be placed relative to the center of the screen.
(a) the_turtle.setposition(0, 0)
(b) the_turtle.setposition(2100, 0)
(c) the_turtle.setposition(250, 0)
(d) the_turtle.setposition(0, 250)

7. For the following method calls on turtle the_turtle, describe the shape that will be drawn. the_turtle.penup()
the_turtle.setposition(2100, 0)
the_turtle.pendown()
the_turtle.setposition(100, 0)
the_turtle.setposition(100, 50)
the_turtle.setposition(2100, 50)
the_turtle.setposition(2100, 0)

8. What color line will be drawn in the following?
turtle.colormode(255)
the_turtle.pencolor(128, 0, 0)
the_turtle.pendown()
the_turtle.forward(100)

9. What will be displayed by the following turtle actions?
the_turtle.pendown()
the_turtle.showturtle()

```
the_turtle.forward(25)
the_turtle.penup()
the_turtle.hide_turtle()
the_turtle.forward(25)
the_turtle.pendown()
the_turtle.showturtle()
the_turtle.forward(25)
```

PYTHON PROGRAMMING EXERCISES

P1. Give a set of instructions for controlling the turtle to draw a line from the top-left corner of the screen to the bottom-right corner, and from the top-right corner to the bottom-left corner, thereby making a big X on the screen. There should be no other lines drawn on the screen.

P2. Using relative positioning, give a set of instructions for controlling the turtle to draw an isosceles triangle on the screen (that is, a triangle with two equal-length sides).
P3. Give a set of instructions for controlling the turtle to draw the letter W using relative positioning.
P4. Give a set of instructions for controlling the turtle to create three concentric circles, each of different color and line width.
P5. Give a set of instructions that sets the turtle to an actual turtle shape, and moves it from the bottom of the screen towards the top, getting smaller as it moves along.
P6. Give a set of instructions that moves the turtle with an actual turtle shape from the bottom of the screen toward the top, changing its fill color when it crosses the x axis of the grid coordinates.
P7. Give a set of instructions to create your own polygon shape and create an interesting design with it.

Program Modification Problems 245

P8. Give a set of instructions so that the turtle initially moves slowly around the edge of the screen, then moves faster and faster as it goes around.
P9. Give a set of instructions to create two turtle objects each with circle shape that move to various locations of the turtle screen, each stamping their circle shape of varying sizes and colors.

PROGRAM MODIFICATION PROBLEMS

M1. Bouncing Balls with Color
Modify the bouncing balls simulation program so that exactly three balls are created, each with a different color.

M2. Bouncing Balls with Changing Color
Modify the bouncing balls simulation program so that each time a ball hits an edge of the turtle graphics screen, it changes color.

M3. Bouncing Balls with Trailing Lines
Modify the bouncing balls simulation program so that a trail is left on the screen of each ball's path.

M4. Bouncing Ball Chase
Modify the bouncing balls simulation program so that there are exactly three balls generated, with the first ball started in a random direction (heading), and the other two balls following it closely behind.

M5. Horse Racing Program: Multiple Races and Score Keeping
Modify the Horse Racing program so that the user can continue to play another race without having to rerun the program. Also, the cumulative wins of all the horses should be displayed in the shell window. This allows the user to see if some horses are a more "winning" horse than others.

M6. Horse Racing Program: Handicap Racing
Modify the Horse Racing program so that the user can assign a handicap to one or more horses on a scale of 1 to 5. (A "handicap" in racing is a means of giving advantage to less competitive horses over more competitive ones.) If a horse is assigned a handicap of 1, it should move ahead one-fifth farther than usual. A handicap of 2 would increase its move by two-fifths, and so forth. The list of handicaps should be displayed in the shell window each time before the race begins.

M7. Horse Racing Program: Pari-mutuel Betting
Modify the Horse Racing program to allow individuals to enter their name to "register themselves" to place bets. The program should be modified so that races can be consecutively run without having to restart the program. Before each race, bets can be placed by registered players. Each bet is for which horse will win. The payout will be based on the rules of *pari-mutuel betting* described below. The amount of money gained or lost by each registered player should be constantly displayed in the shell.

Example of pari-mutuel betting
Each horse has a certain amount of money wagered on it (assuming eight horses):
12345678 $30.00 $70.00 $12.00 $55.00 $110.00 $47.00 $150.00 $40.00

Thus, the total *pool* of money on this particular wagering event is $514.00. Following the start of the event, no more wagers are accepted. The event is decided and the winning outcome is determined to be outcome 4 with $55.00 wagered. The *payout* is now calculated. First, the *commission* or *take* for the wagering company is deducted from the pool. For example, with a commission rate of 14.25% the pool is: $514 3 (1 2 0.1425) 5 $440.76. This remaining amount in the pool is now distributed to those who wagered on outcome 4: $440.76 / $55 ≈ $8 per $1 wagered. This payout includes the $1 wagered plus an additional $7 profit. Thus, the odds of outcome 4 are 7-to-1.

Wikipedia contributors. "Parimutuel betting." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, May 7, 2011. Web, May 11, 2011.
PROGRAM DEVELOPMENT PROBLEMS

D1. Drunkard's Walk
A *random walk* is a trajectory taken by a sequence of random steps. Random walks can be used to model the travel of molecules, the path that animals take when looking for food, and financial fluctuations, for example. A specific form of random walk is called the "Drunkard's Walk." A (drunken) man tries to find his way home. He does so by making a random choice at each street intersection of which of the four paths to take: continue in the same direction; go back from the direction he came; turn left; or turn right. Thus, the man is traveling the same distance after each choice of direction (one city block). Implement and test a Python program using turtle graphics to display random walks. Select an appropriate number of pixels as the length of a city block.

D2. Name Reversal
Implement and test a Python program using turtle graphics to allow the user to enter their first name, and have it displayed in the turtle window as a reverse mirror image.

D3. Battleship Game Visualization
Implement and test a Python program using turtle graphics to provide a

visualization for the game of Battleship discussed in Program Development problem D3 in Chapter 4.

Modular Design CHAPTER 7

Until now, we have looked at programs comprised of a set of individual functions. In complex software systems, however, programs are organized at a higher level as a set of modules, each module containing a set of functions (or objects). Modules, like functions, are a fundamental building block in software development.

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Explain the use of modular design in software development ♦ Explain the specification of modules
♦ Explain the process of top-down design
♦ Describe the concept of a stack
♦ Differentiate between unit testing and integration testing ♦ Become familiar with the use of docstrings in Python ♦ Explain the use of modules and namespaces in Python ♦ Describe and use the different forms of module import in Python ♦ Develop well-specified Python programs using top-down design

CHAPTER CONTENTS

Motivation
Fundamental Concepts
7.1 Modules
7.2 Top-Down Design
7.3 Python Modules
Computational Problem Solving
7.4 Calendar Year Program (function version)

MOTIVATION

Software systems are some of the most complex entities ever created. Web browsers and operating systems, for example, contain as many as 50–100 million lines of code. Developing programs of such size and complexity can easily take 10,000 person-years of effort to develop. It is natural, therefore, to find ways to divide the task of software development among various individuals (or groups of individuals).

We can make complex systems more manageable by designing them as a set of subsystems, or modules. For example, NASA's space shuttle (Figure 7-1) is one of the most complex systems ever engineered, containing more than 2.5 million parts. The major components are the orbiter vehicle, a large external liquid-fuel tank, and two solid rocket boosters. The orbiter vehicle itself is composed of several subsystems (e.g., the communications system), which in turn may be composed of sub-subsystems (e.g., the data network system), as shown in Figure 7-2.

FIGURE 7-1 Space Shuttle

FIGURE 7-2 Modular Design of the NASA Space Shuttle

Although many of the technicians involved with the shuttle may understand its overall design, only a few select individuals need to understand or be involved with the detailed design, implementation, and testing of any specific subsystem. The same is true for the modular design of software. In this chapter, we look at the issue of modular program design.

FUNDAMENTAL CONCEPTS

7.1 Modules

7.1.1 What Is a Module?

An important aspect of well-designed software is that programs are designed as a

collection of modules. The term “module,” broadly speaking, refers to the design and/or implementation of

specific functionality to be incorporated into a program. While an individual function may be considered a module, modules generally consists of a collection of functions (or other entities). The Python turtle module is an example of a software module. The use of modules has a number of advantages as shown in Figure 7-3.

- SOFTWARE DESIGN
  - provides a means for the development of well-designed programs
- SOFTWARE DEVELOPMENT
  - provides a natural means of dividing up programming tasks
  - provides a means for the reuse of program code
- SOFTWARE TESTING
  - provides a means of separately testing parts of a program
  - provides a means of integrating parts of a program during testing
- SOFTWARE MODIFICATION AND MAINTENANCE
  - facilitates the modification of specific program functionalities

FIGURE 7-3 Advantages of Modular Programming

Modular design allows large programs to be broken down into manageable size parts, in which each part (module) provides a clearly specified capability. It aids the software development process by providing an effective way of separating programming tasks among various individuals or teams. It allows modules to be individually developed and tested, and eventually integrated as a part of a complete system. Finally, modular design facilitates program modification since the code responsible for a given aspect of the software is contained within specific modules, and not distributed throughout the program.

The term “**module**” refers to the design and/or implementation of specific functionality to be incorporated into a program.

7.1.2 Module Specification

Every module needs to provide a *specification* of how it is to be used. This is referred to as the module’s **interface**. Any program code making use of a particular module is referred to as a **client** of the module. A module’s

specification should be sufficiently *clear* and *complete* so that its clients can effectively utilize it. For example, numPrimes is a function that returns the number of primes in a given integer range, as shown in Figure 7-4.

```
def numPrimes(start, end):

    """ Returns the number of primes between start and end. """
```

FIGURE 7-4 docstring Specification

The function's specification is provided by the line immediately following the function header, called a *docstring* in Python. A **docstring** is a string literal denoted by triple quotes given as the first line of certain program elements. The docstring of a particular program element can be displayed by use of the __doc__ extension,

... print(numPrimes.__doc__)
Returns the number of primes between start and end.

This provides a convenient way for discovering how to use a particular function without having to look at the function definition itself. Some software development tools also make use of docstrings. Let's consider how complete this specification is for this function. At first look it may seem sufficient. However, it does not answer whether the number of primes returned includes the endpoints of the range or not. Also, it is not clear what will happen if the function is called with a first argument (start) greater than the second (end). Thus, a more complete specification is needed. This is given in Figure 7-5.

```
def numprimes(start, end):

    """ Returns the number of primes between start and end, inclusive.

        Returns -1 if start is greater than end.
    """
```

FIGURE 7-5 A More Complete docstring Specification

This is now a reasonable specification of the function. This docstring follows the Python convention of putting a blank line after the first line of the docstring, which should be an overall description of what the function does, followed by an arbitrary number of lines providing additional details. These additional lines must be indented at the same level, as shown in the figure. Appropriate use of

this function by a client is given below.

```
.
.
first_num 5 int(input('Enter the start of the range: '))
second_num 5 int(input('Enter the end of the range: '))

result 5 numprimes(first_num, second_num)
if result 55 21:
print('* Invalid range entered *')
else:
print('The number of primes between', first_num, 'and', second_num, 'is', result)
```

In this example, the user inputs a start and end value for the range of integers to check. Since the user may inappropriately enter a start value greater than the value of the end value, a check is made for a returned value of 21 after the call to numprimes. If 21 is found, then an error message is output; otherwise, the result is displayed.

There are potential problems when returning both a computed result and an error result as a function's return value. First, there is no guarantee that the client will perform the necessary check for the special error value. Thus, an incorrect result may be displayed to the user,

The number of primes between 100 and 1 is 21

Second, there may not be a special value that *can* be returned for error reporting. For example, if there is a function meant to return any integer value, including negative numbers, there is no special integer value that can be returned. We will see a better means of error reporting by a function when we discuss exception handling in Chapter 8.

A module's **interface** is a specification of what it provides and how it is to be used. Any program code making use of a given module is called a **client** of the module. A **docstring** is a string literal denoted by triple quotes used in Python for providing the specification of certain program elements.

Self-Test Questions

1. Which of the following is not an advantage in the use of modules in software

development? **(a)** Provides a natural means of dividing up programming tasks.
(b) Provides a means of reducing the size of a program.
(c) Provides a means for the reuse of program code.
(d) Provides a means of separately testing individual parts of a program.
(e) Provides a means of integrating parts of a program during testing.
(f) Facilitates the modification of specific program functionalities.

2. A specification of how a particular module is used is called the module's _____.
3. Program code that makes use of a given module is called a _____ of the module.

4. Indicate which of the following are true. A docstring in Python is
(a) A string literal denoted by triple or double quotes.
(b) A means of providing specification for certain program elements in Python.
(c) A string literal that may span more than one line.

ANSWERS: 1. (b), 2. interface, 3. client, 4. (b), (c)

7.2 Top-Down Design

One method of deriving a modular design is called **top-down design**. In this approach, the overall design of a system is developed first, deferring the specification of more detailed aspects of the design until later steps. We next consider a modular design using a top-down approach for the calendar year program from Chapter 4.

Top-down design is an approach for deriving a modular design in which the overall design of a system is developed first, deferring the specification of more detailed aspects of the design until later steps.

7.2.1 Developing a Modular Design of the Calendar Year Program

We will develop a modular design for the calendar year program from Chapter 4 (implemented there without the use of functions) using a top-down design approach. The three overall steps of the program are getting the requested year from the user, creating the calendar year structure, and displaying the year. This is depicted in Figure 7-6.

FIGURE 7-6 First Stage of a Modular Design of the Calendar Year Program

We then consider whether any of these modules needs to be further broken down. Making such a decision is more of an art than a science. The goal of modular design is that each module provides clearly defined functionality, which collectively provide all of the required functionality of the program. Modules get year and display calendar year are not complex enough to require further breakdown. Module construct calendar year, on the other hand, is where most of the work is done, and is therefore further broken down. Figure 7-7 contains the modules of this next design step.

FIGURE 7-7 Second Stage of Modular Design of a Calendar Year Program

In order to construct a calendar year, it must be determined whether the year is a leap year, what day of the week January 1st of that year falls on, and how many days are in each month (accounting for leap years). Thus, modules leap year, day of week January1, and num days in month are added as *submodules* of module construct calendar year. The calendar month for each of the twelve months must then be individually constructed, handled by module construct calendar month.

The goal of top-down design is that each module provides clearly defined functionality, which collectively provide all of the required functionality of the program.

7.2.2 Specification of the Calendar Year Program Modules

The modular design of the calendar year program provides a high-level view of the program. However, there are many issues yet to resolve in the design. Since each module is to be implemented as a function, we need to specify the details of each function. For example, for each function it needs to be decided if it is a value-returning function or a non-value-returning function; what parameters it will take; and what results it will produce. We give such a specification in Figure 7-8 using Python docstrings.

This stage of the design provides sufficient detail from which to implement the program. The main module provides the overall construction of the program. It simply displays the program

```
def getYear():

    """ Returns an integer value between 1800-2099, inclusive, or  -1."""

def leapYear(year):

    """ Returns True if provided year a leap year, otherwise returns False."""

def dayOfWeekJan1(year, leap_year):

    """ Returns the day of the week for January 1 of the provided year.

        year must be between 1800 and 2099. leap_year must be True if
        year a leap year, and False otherwise.
    """

def numDaysInMonth(month_num, leap_year):

    """ Returns the number of days in a given month.

        month_num must be in the range 1-12, inclusive.
        leap_year must be True if month in a leap year, otherwise False.
    """

def constructCalMonth(month_num, first_day_of_month, num_days_in_month):

    """ Returns a formatted calendar month for display on the screen.

        month_num must be in the range 1-12, inclusive.
        first_day_of_month must be in the range 0-6 (1-Sun, 2-Mon, ..., 0-Sat).

        Returns a list of strings of the form,
        [month_name, week1, week2, week3, week4, week5, week6].
    """

def constructCalYear(year):

    """ Returns a formatted calendar year for display on the screen.

        year must be in the range 1800-2099, inclusive.
        Returns a list of list of strings of the form,

        [year, month1, month2, month3, …, month12].
    """

def displayCalendar(calendar_year):

    """ Displays the provided calendar_year on the screen three months across.

        Provided calendar year should be in the form of a list of twelve sublists, in which each
        sublist is of the form [monthname, week1, week2, …, ]
    """
```

FIGURE 7-8 Calendar Year Module Specification (*Continued*)

```
# main
terminate = False

print('This program will display a calendar year for a given year')

while not terminate:
    year = getYear()

    if year ==  -1:
       terminate = True
    else:
        calendar_year = constructCalYear(year)
        displayCalendar(calendar_year)
```

FIGURE 7-8 Calendar Year Module Specification

greeting, calls module getYear to get the year from the user, calls module constructCalYear to construct the year, and finally calls module displayCalendar to display the calendar year. The only detail that the main module is concerned with is allowing the user to keep displaying another calendar year, or enter 21 to terminate the program. This is controlled by Boolean variable terminate.

The first module called, getYear, returns the integer value entered by the user. The module's specification indicates that it returns an integer value between 1800 and 2099, inclusive; or 21 (if the user decides to terminate the program). Therefore, it is the responsibility of the module to ensure that no other value is returned. This relieves the main module of having to check for bad input.

The next module that the main module uses is module constructCalYear. This module returns a list of twelve sublists, one for each month, in which each sublist begins with the name of the month (as a string value), followed by each of the weeks in the month, each week formatted as a single string. The module's specification indicates that it is to be passed a year between 1800 and 2099, inclusive. Therefore, any year given to it that is outside that range violates its condition for use, and therefore the results are not guaranteed.

The last module called from the main module is module displayCalendar. It is given a constructed calendar year, as constructed by module constructCalMonth. Based on the modular design of the calendar year program, constructCalYear is the only module relying on the use of submodules, specifically modules leapYear, dayOfWeekJan1, numDaysInMonth, and constructCal. The leapYear module determines whether a given a year is a leap year or not, returning a

Boolean result. Module dayOfWeekJan1 returns the day of the week for January 1st of the provided year. Boolean value leap_year must also be provided to the module, needed in the day of the week algorithm on which the module is based. Module numDaysInMonth must be passed an integer in the range 1–12, as well as a Boolean value for leap_year. This is so that the module can determine the number of days in the month for the month of February. Finally, constructCalMonth is given a month number, the day of the week of the first day of the month (1-Sun, 2-Mon, . . ., 0-Sat) and the number of days in the month. With this information, module constructCalYear can construct the calendar list and its sublists to be displayed.
Finally, module displayCalendarMonth is given a formatted calendar year. Its job is to display the calendar year three months across.
This more detailed modular design provides the details of how each module is to be incorporated into a complete program. We will discuss the implementation of this design at the end of the chapter.

Self-Test Questions

1. In top-down design (select one),
(a) The details of a program design are addressed before the overall design. **(b)** The overall design of a program is addressed before the details.

2. All modular designs are a result of a top-down design process. (TRUE/FALSE) **3.** In top-down design, every module is broken down into the same number of submodules. (TRUE/FALSE)

4. Which of the following advantages of modular design apply to the design of the calendar program.
(a) Provides a means for the development of well-designed programs.
(b) Provides a natural means of dividing up programming tasks.
(c) Provides a means of separately testing individual parts of a program.

ANSWERS: 1. (b), 2. False, 3. False, 4. (a), (b), (c)

7.3 Python Modules
7.3.1 What Is a Python Module?

A Python module is a file containing Python definitions and statements. When a Python file is directly executed, it is considered the *main module* of a program. Main modules are given the special name __main__. Main modules provide the

basis for a complete Python program. They may *import* (include) any number of other modules (and each of those modules import other modules, etc.). Main modules are not meant to be imported into other modules.

As with the main module, imported modules may contain a set of statements. The statements of imported modules are executed only once, the first time that the module is imported. The purpose of these statements is to perform any initialization needed for the members of the imported module. The Python Standard Library contains a set of predefined *Standard (built-in) modules*. We have in fact seen some of these modules already, such as the math and random Standard Library modules.

Python modules provide all the benefits of modular software design we have discussed. By convention, modules are named using all lower case letters and optional underscore characters. We will look more closely at Python modules in the next section.

LET'S TRY IT Create a Python module by entering the following in a file name simple.py. Then execute the instructions in the Python shell as shown and observe the results. # module simple
print('module simple loaded')

def func1():
print('func1 called') ... import simple ???
... simple.func1() ???

def func2():
print('func2 called') ... simple.func2() ???

A **Python module** is a file containing Python definitions and statements. The Python Standard Library contains a set of predefined **standard (built-in) modules**.

7.3.2 Modules and Namespaces

A **namespace** is a container that provides a named context for a set of identifiers. Namespaces enable programs to avoid potential *name clashes* by associating each identifier with the namespace from which it originates. In software development, a **name clash** is when two otherwise distinct entities with the same name become part of the same scope. Name clashes can occur, for example, if two or more Python modules contain identifiers with the same name

and are imported into the same program, as shown in Figure 7-9.

```
# module1

def double(lst):

    """Returns a new list with each
       number doubled, for example,
       [1, 2, 3] returned as [2, 4, 6]
    """
```

```
# module2

def double(lst):

    """Returns a new list with each
       number duplicated, for example,
       [1, 2, 3] returned as
       [(1, 1), (2, 2), (3, 3)]
    """
```

```
import module1
import module2

# main
   .
  num_list = [3, 8, 14]
  result = double(num_list)
   .
```

ambiguous reference for identifier `double`

FIGURE 7-9 Example Name Clash of Imported Functions

In this example, module1 and module2 are imported into the same program. Each module contains an identifier named double, which return very different results. When the function call double(num_list) is executed in main, there is a name clash. Thus, it cannot be determined which of these two functions should be called. Namespaces provide a means for resolving such problems.

In Python, each module has its own namespace. This includes the names of all items in the module, including functions and global variables—variables defined within the module and outside the scope of any of its functions. Thus, two instances of identifier double, each defined in their own module, are distinguished by being *fully qualified* with the name of the module in which each is defined: module1.double and module2.double. Figure 7-10 illustrates the use of fully qualified identifiers for calls to function double.

The use of namespaces to resolve problems associated with duplicate naming is not restricted to computer programming. In fact, it occurs in everyday situations. Imagine, for instance, that you

```
import module1
import module2

# main

   ans1 = module1.double(...)

   ans2 = module2.double(...)
```

references function double from module1's namespace

references function double from module2's namespace

FIGURE 7-10 Example Use of Fully Qualified Function Names

run into a friend who tells you that "Paul is getting married." In fact, you have two friends in common named Paul. Because you are not certain which Paul your friend is referring to, you may respond "Paul from back home, or Paul from the dorm?" In this case, you are asking your friend to respond with a fully qualified name to resolve the ambiguity: "home:Paul" vs. "dorm:Paul." Next, we look at different ways that Python modules can be imported.

LET'S TRY IT

Enter each of the following functions in their own modules named mod1.py and mod2.py. Enter and execute the following and observe the results.

```
# mod1
def average(lst):
print('average of mod1 called')

... import mod1, mod2
... mod1.average([10, 20, 30]) ???
... mod2.average([10, 20, 30])

# mod2
def average(lst):
print('average of mod2 called')

???
... average([10, 20, 30]) ???
```

A **namespace** provides a context for a set of identifiers. Every module in Python has its own namespace. A **name clash** is when two otherwise distinct entities with the same identifier become part of the same scope.

7.3.3 Importing Modules

In Python, the **main module** of any program is the first ("top-level") module executed. When working interactively in the Python shell, the Python interpreter functions as the main module, containing the *global namespace*. The namespace is reset every time the interpreter is started (or when selecting Shell → Restart Shell). Next we look at various means of importing modules in Python. (We note that module __builtins__ is automatically imported in Python programs, providing all the built-in constants, functions, and classes.)

In Python, the **main module** of any program is identified as the first ("top-level") module executed.
The "import modulename" Form of Import

When using the import *modulename* form of import, the namespace of the imported module becomes *available to*, but not *part of*, the importing module. Identifiers of the imported module, therefore, must be fully qualified (prefixed with the module's name) when accessed. Using this form of import prevents any possibility of a name clash. Thus, as we have seen, if two modules, module1 and module2, both have the same identifier, identifier1, then module1.identifier1 denotes the entity of the first module and module2.identifier1 denotes the entity of the second module.

LET'S TRY IT
Enter the following into the Python shell and observe the results.

... factorial(5) ... import math ??? ... factorial(5)

???
... math.factorial(5)
??? ... math.factorial(5)

???
With the import *modulename* form of import in Python, the namespace of the imported module becomes available to, but does not become part of, the namespace of the importing module.
The "from-import" Form of Import
Python also provides an alternate import statement of the form
from *modulename* **import** *something*
where *something* can be a list of identifiers, a single renamed identifier, or an

asterisk, as shown below,

(a) **from** *modulename* **import** func1, func2 (b)**from** *modulename* **import** func1 **as** new_func1 (c)**from** *modulename* **import** *

In example (a), only identifiers func1 and func2 are imported. In example (b), only identifier func1 is imported, renamed as new_func1 in the importing module. Finally, in example (c), *all* of the identifiers are imported, except for those that begin with two underscore characters, which are meant to be private in the module, which will be discussed later.

There is a fundamental difference between the from *modulename* import and import *modulename* forms of import in Python. When using import *modulename*, the namespace of the imported module does not become part of the namespace of the importing module, as mentioned. Therefore, identifiers of the imported module must be fully qualified (e.g., *modulename*. func1) in the importing module. In contrast, when using from-import, the imported module's namespace *becomes part of* the importing module's namespace. Thus, imported identifiers are referenced without being fully qualified (e.g., func1).

The from *modulename* import func1 as new_func1 form of import is used when identifiers in the imported module's namespace are known to be identical to identifiers of the importing module. In such cases, the renamed imported function can be used without needing to be fully qualified. Finally, using the from *modulename* import * form of import in example (c), although convenient, makes name clashes more likely. This is because the names of the imported identifiers are not explicitly listed in the import statement, creating a greater chance that the programmer will unintentionally define an identifier with the same name as in the importing module. And since the from-import form of import allows imported identifiers to be accessed without being fully qualified, it is unclear in the importing module where these identifiers come from.
We provide an example of this in Figure 7-11.

```
# module somemodule

def func1(n):
  return n * 10

def func2(n1, n2):
  return n1 * n2
```

```
# ---- main

from somemodule import *

# NAMESPACES AND FUNCTION USE

def func2(n):
    return n * n            # definition of func2 masks imported func2

print(func1(8))             # outputs 80 (func1 of somemodule called)
print(somemodule.func1(8))  # NameError: name 'somemodule' is not defined

print(func2(5))             # outputs 25 (func2 of MAIN called)
print(func2(3, 8))          # TypeError: func2() takes exactly 1 argument

print(somemodule.func2(3, 8)) # NameError: name 'somemodule' is not
                                defined
```

FIGURE 7-11 Example Use of from-import Form of Import

Module somemodule contains functions func1 and func2. Since somemodule is imported with from somemodule import *, identifiers func1 and func2 become part of the main module's namespace. However, since the module's namespace already contains identifier func2 (denoting the function defined there), access to func2 of somemodule is *masked*, and therefore is inaccessible. Using the fully qualified form somemodule.func2 does not work either, since somemodule is not part of the imported namespace for this form of import.

Finally, it is recommended Python style that standard modules be imported before the programmer-defined ones, with each section of imports separated by a blank line as shown below.
import standardmodule1 # standard modules
import standardmodule2
import somemodule1 # programmer-defined modules import somemodule2
LET'S TRY IT Enter the following into the Python shell and observe the results.

... from math import factorial ... from math import factorial as fact ... factorial(5)
... fact(5)

??? ???

... def factorial(n):
print('my factorial') ... factorial(5)
??? ??? ... def factorial(n):
print('my factorial') ... factorial(5)
... math.factorial(5) ... fact(5) ??? ???

With the from-import form of import, imported identifiers become part of the importing module's namespace. Because of the possibility of name clashes, import *modulename* is the preferred form of import in Python.

Module Private Variables

In Python, all identifiers in a module are "public"—that is, accessible by any other module that imports it. Sometimes, however, entities (variables, functions, etc.) in a module are meant to be "private"—used within the module, but not meant to be accessed from outside it.

Python does not provide any means for preventing access to variables or other entities meant to be private. Instead, there is a convention that names beginning with two underscores (_) are intended to be private. Such entities, therefore, *should not* be accessed. It does not mean that they *cannot* be accessed, however. There is one situation in which access to private variables is restricted. When the from *modulename* import * form of import is used to import *all* the identifiers of a module's namespace, names beginning with double underscores are not imported. Thus, such entities become inaccessible from within the importing module.

In Python, all the variables in a module are "public," with the convention that variables beginning with an two underscores are intended to be private.
7.3.4 Module Loading and Execution

Each imported module of a Python program needs to be located and loaded into memory. Python first searches for modules in the current directory. If the module is not found, it searches the directories specified in the PYTHONPATH environment variable. If the module is still not found (or PYTHONPATH is not defined), a Python installation-specific path is searched (e.g., C:\Python32\Lib). If the program still does not find the module, an error (ImportError exception) is reported. For our purposes, all of the modules of a program will be kept in the

same directory. However, if you wish to develop a module made available to other programs, then the module can be saved in your own Python modules directory specified in the PYTHONPATH, or stored in the particular Python installation Lib directory. When a module is loaded, a compiled version of the module with file extension .pyc is automatically produced. Then, the next time that the module is imported, the compiled .pyc file is loaded, rather than the .py file, to save the time of recompiling. A new compiled version of a module is automatically produced whenever the compiled version is out of date with the source code version of the module when loading, based on the dates that the files were created/modified.

Built-in Function **dir()**

Built-in function dir() is very useful for monitoring the items in the namespace of the main module for programs executing in the Python shell. For example, the following gives the namespace of a newly started shell,

```
... dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

The following shows the namespace after importing and defining variables,

```
... import random
... n 5 10
... dir()
['__builtins__', '__doc__', '__name__', '__package__', 'n', 'random']
```

Selecting Shell→ Restart Shell (Ctrl-F6) in the shell resets the namespace,

```
(after Restart Shell selected)
... dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

LET’S TRY IT Create the following Python module named simplemodule, import it, and call function displayGreeting as shown from the Python shell and observe the results.

```
# simplemodule
def displayGreeting():
print('Hello World!')
... import simplemodule
... simplemodule.displayGreeting()
```

Modify module simplemodule to display 'Hey there world!', import and again execute function displayGreeting as shown. Observe the results.

```
... import simplemodule
... simplemodule.displayGreeting()
```

Finally, reload the module as shown and again call function displayGreeting.

```
... reload(simplemodule)
... simplemodule.displayGreeting() ???
```

When a module is loaded, a compiled version of the module with file extension .pyc is automatically produced. When using the Python shell, an updated module can be forced to be reloaded and recompiled by use of the reload() function.

7.3.5 Local, Global, and Built-in Namespaces in Python

During a Python program's execution, there are as many as three namespaces that are referenced ("active")—the built-in namespace, the global namespace, and the local namespace. The **built-in namespace** contains the names of all the built-in functions, constants, and so on, in Python. The **global namespace** contains the identifiers of the currently executing module. And the **local namespace** is the namespace of the currently executing function (if any).

When Python looks for an identifier, it first searches the local namespace (if defined), then the global namespace, and finally the built-in namespace. Thus, if the same identifier is defined in more than one of these namespaces, it becomes masked, as depicted in Figure 7-12.

FIGURE 7-12 Local, Global, and Built-in Namespaces of Python

Functions sum and max are built-in functions in Python, and thus in the built-in namespace. Built-in function sum returns the sum of a sequence (list or tuple) or integers. Built-in function max returns the largest value of a string, list, tuple, and other types.

In the module (and thus part of the global namespace) is defined another function named max. This programmer-defined function returns the index of the largest value from an ordered collection of items, not the value itself as built-in function max is designed to do. This is demonstrated below.

max([4, 2, 7, 1, 9, 6]) → 9 (built-in function max)
max([4, 2, 7, 1, 9, 6]) → 4 (programmer-defined function max)

Which specific functions are called from within somefunction depends on where the functions are defined. Function sum, for example, is not defined within the global namespace. Therefore, built-in function sum of the built-in namespace is called. The call to function max, on the other hand, does not access the built-in function max; rather, it calls function max of the more closely defined global namespace. This demonstrates how issues of scope, if not clearly considered, can result in subtle and unexpected program errors. Consider the example program in Figure 7-13.

```
# grade_calc module

def max(grades):
    largest = 0

    for k in grades:
        if k > 100:
            largest = 100
        elif k > largest:
            largest = k

    return largest

def grades_highlow(grades):
    return (min(grades), max(grades))
```

```
# classgrades (main module)

from grade_calc import *

class_grades = [86, 72, 94, 102, 89, 76, 96]

low_grade, high_grade = grades_highlow(class_grades)
print('Highest adjusted grade on the exam was', high_grade)
print('Lowest grade on the exam was', low_grade)

print('Actual highest grade on exam was', max(class_grades))
```

FIGURE 7-13 Inadvertent Masking of Identifier in the Built-in Namespace

This program is meant to read in the exam grades of a class. (The grades are hard-coded here for the sake of an example.) The main module imports module grade_calc that contains function grades_ highlow, which returns as a tuple the highest grade (with extra credit grades over 100 returned as 100) and lowest grade in a list of grades. Upon executing this program, we find the following results,

Highest adjusted grade on the exam was 100
Lowest grade on the exam was 72
The actual highest grade on the exam was 100
...

For the list of grades 86, 72, 94, 102, 89, 76, 96, the high and low grades of 72

and 100 is correct (counting the grade 102 as a grade of 100). Then the program is to display the actual highest grade of 102. However, a grade of 100 is displayed instead.

The problem is that the grade_calc module was imported using from grade_calc import *. Thus, all of the entities of the module were imported, including the defined max function. This function returns a “truncated” maximum grade of 100 from a list of grades, used as a supporting function for function grades_highlow. However, since it is defined in the global (module) namespace of the classgrades program, that definition of function max masks the built-in max function of the built-in namespace. Thus, when called from within the classgrades program, it also produces a truncated highest grade, thus returning the actual highest grade of 100 instead of 102.

This example shows the care that must be taken in the use and naming of global identifiers, especially with the from-import * form of import. Note that if function max were named as a private member of the module, __max, then it would not have been imported into the main module and the actual highest grade displayed would have been correct.

Enter the following in the Python shell: ... sum([1, 2, 3])
???

LET’S TRY IT
Create a file with the following module:

# module max_test_module
def test_max():
print 'max 5', max([1, 2, 3]) ... def sum(n1, n2, n3): total 5 n1 1 n2 1 n3 return total
... sum([1, 2, 3])
???
... sum(1, 2, 3)
???

Create and execute the following program: import max_test_module

def max():
print('max:local namespace called') print(max_test_module.test_max())
At any given point in a Python program’s execution, there are three possible

namespaces referenced (“active”)—the **built-in namespace**, the **global namespace**, and the **local namespace**.

7.3.6 A Programmer-Defined Stack Module

In order to demonstrate the development of a programmer-defined module, we present an example stack module. A **stack** is a very useful mechanism in computer science. Stacks are used to temporarily store and retrieve data. They have the property that the last item placed on the stack is the first to be retrieved. This is referred to as LIFO—“last in, first out.” A stack can be viewed as a list that can be accessed only at one end, as depicted in Figure 7-14.

FIGURE 7-14 Stack Mechanism

In this example, three items are *pushed* on the stack, denoted by A, B, and C. First, item A is pushed, followed by item B, and then item C. After the three items have been placed on the stack, the only item that can be accessed or removed is item C, located at the *top of stack*. When C is retrieved, it is said to be *popped* from the stack, leaving item B as the top of stack. Once item B is popped, item A becomes the top of stack. Finally, when item A is popped, the stack becomes empty. It is an error to attempt to pop an empty stack.

In Figure 7-15 is a Python module containing a set of functions that implements this stack behavior. For demonstration purposes, the program displays and pushes the values 1 through 4 on the stack. It then displays the numbers popped off the stack, retrieved in the reverse order that they were pushed.

The stack module consists of five functions—getStack, isEmpty, top, push, and

pop. The stack is implemented as a list. Only the last element in the list is accessed—that is where

```
1  # stack Module
2
3  def getStack():
4
5      """Creates and returns an empty stack."""
6
7      return []
8
9  def isEmpty(s):
10
11     """Returns True if stack empty, otherwise returns False. """
12
13     if s == []:
14         return True
15     else:
16         return False
17
18 def top(s):
19
20     """Returns value of the top item of stack, if stack not empty.
21        Otherwise, returns None.
22     """
23
24     if isEmpty(s):
25         return None
26     else:
27         return s[len(s) - 1]
28
29 def push(s, item):
30
31     """Pushes item on the top of stack. """
32
33     s.append(item)
34
35 def pop(s):
36
37     """Returns top of stack if stack not empty. Otherwise, returns None."""
38
39     if isEmpty(s):
40         return None
41     else:
42         item = s[len(s) - 1]
43         del s[len(s) - 1]
44         return item
```

FIGURE 7-15 Programmer-Defined Stack Module

all items are “pushed” and “popped” from. Thus, the end of the list logically functions as the “top” of stack.

Function getStack**(lines 3–7)** creates and returns a new empty stack as an empty list. Function isEmpty **(lines 9–16)** returns whether a stack is empty or not (by checking if an empty list). Function top (**lines 18–27**) returns the top item of a stack without removing it. Functions push **(lines 29–33)** and pop **(**

lines 35–44) provide the essential stack operations. The push function pushes an item on the stack by appending it to the end of the list. The pop function removes the item from the top of stack by retrieving the last element of the list, and then deleting it. If either pop or top are called on an empty stack, the special value None is returned.

```
# main
1  import stack
2
3  mystack = stack.getStack()
4
5  for item in range(1, 5):
6      stack.push(mystack, item)
7      print('Pushing', item, 'on stack')
8
9  while not stack.isEmpty(mystack):
10     item = stack.pop(mystack)
11     print('Popping', item,'from stack')
```

FIGURE 7-16 Demonstration of the Programmer-Defined Stack Module

The small program in Figure 7-16 demonstrates the use of the stack module. First, a new stack is created by assigning variable mystack to the result of the call to function getStack **(line 3)**. Even though a list is returned, it is intended to be more specifically a stack type. Therefore, only the stack-related functions should be used with this variable—the list should not be directly accessed, otherwise the stack may become corrupted. Then the values 1 through 4 are pushed on the stack, and popped in the reverse order,

Pushing 1 on stack Pushing 2 on stack Pushing 3 on stack Pushing 4 on stack Popping 4 from stack Popping 3 from stack Popping 2 from stack Popping 1 from stack

Ensuring that only the provided functions can be used on the stack represents a fundamental advantage of objects. Since an object consists of data and methods (routines), only the methods of the object can be used to access and alter the data. Thus, the stack type is best implemented as an object, as all other values in Python are. We will see how do to this after the introduction of object-oriented programming in Chapter 10.

7.3.7 Let’s Apply It—A Palindrome Checker Program

The program in Figure 7-18 determines if a given string is a palindrome. A

palindrome is something that reads the same forwards and backwards. For example, the words “level” and “radar” are palindromes. The program imports the stack module developed in the previous section. This program utilizes the following programming feature:

Programmer-defined module ➤
Example execution of the program is given in Figure 7-17.

```
Program execution ...


This program can determine if a given string is a palindrome

(Enter return to exit)
Enter string to check: look
look is NOT a palindrome

Enter string to check: radar
radar is a palindrome

Enter string to check:
>>>
```

FIGURE 7-17 Execution of the Palindrome Checker Program

The stack module is imported on **line 3** of the program. The “import *modulename*” form of import is used. Therefore, each stack function is referenced by stack.*function_name*. **Lines 6–7** displays a simple program welcome. The following lines perform the required initialization for the program. **Line 10**sets char_stack to a new empty stack by call to getStack(). **Line 11** initializes variable empty_string to the empty string. This is used in the program for determining if the user has finished entering all the words to check.

The string to check is input by the user on **line 14**. If the string is of length one, then by definition the string is a palindrome. This special case is handled in **lines 17–18**. Otherwise, the complete string is checked. First, variable palindrome is initialized to True. On **line 24**, variable compare_ length is set to half the length of the input string, using integer division to truncate the length to an equal number of characters. This represents the number of characters from the front of the string (working forward) that must match the number of characters on the rear of the string (working backwards). If there are an odd number of characters, then the middle character has no other character to match against.

On **lines 27–28** the second half of the string chars are pushed character-by-character onto the stack. Then, on lines **31–37** the characters are popped from the stack one by one, returning in the reverse order that they were pushed. Thus, the first character popped (the *last* character pushed on the stack) is compared to the *fi r s t* character of the complete string. This continues until there are no more characters to be checked. If characters are found that do not match, then is_palindrome is set to False (**lines 34–35**) and the while loop terminates. Otherwise, is_palindrome remains True. **Lines 40–43** output whether the input string is a palindrome or not, based on the final value of is_palindrome. **Lines 45–46** prompt the user for another string to enter, and control returns to the top of the while loop.

```python
# Palindrome Checker Program

import stack

# welcome
print('This program can determine if a given string is a palindrome\n')
print('(Enter return to exit)')

# init
char_stack = stack.getStack()
empty_string = ''

# get string from user
chars = input('Enter string to check: ')

while chars != empty_string:
    if len(chars) == 1:
        print('A one letter word is by definition a palindrome\n')
    else:
        # init
        is_palindrome = True

        # to handle strings of odd length
        compare_length = len(chars) // 2

        # push second half of input string on stack
        for k in range(compare_length, len(chars)):
            stack.push(char_stack, chars[k])

        # pop chars and compare to first half of string
        k = 0
        while k < compare_length and is_palindrome:
            ch = stack.pop(char_stack)
            if chars[k].lower() != ch.lower():
                is_palindrome = False

            k = k + 1

        # display results
        if is_palindrome:
            print(chars, 'is a palindrome\n')
        else:
            print(chars, 'is NOT a palindrome\n')

    # get next string from user
    chars = input('Enter string to check: ')
```

FIGURE 7-18 Palindrome Checker Program

It is important to mention that the problem of palindrome checking could be done more efficiently without the use of a stack. A for loop can be used that compares the characters k locations from each end of the given string. Thus, our use of a stack for this problem was for demonstration purposes only. We leave the checking of palindromes by iteration as a chapter exercise.

Self-Test Questions

1. Any initialization code in a Python module is only executed once, the first time that the module is loaded. (TRUE/FALSE)
2. With the "import *moduleName*" form of import, any utilized entities from the imported module must be prefixed with the module name. (TRUE/FALSE)
3. By convention, variables names in a module beginning with two _____ characters are meant to be treated as private variables of the module.
4. When importing modules, all Python Standard Library modules must be imported before any programmer-defined modules, otherwise a runtime error will occur. (TRUE/FALSE)

5. If a particular module is imported more than once in a Python program, the Python interpreter will ensure that the module is only loaded and executed the first time that it is imported. (TRUE/FALSE)

6. The _____ command can be used to force the reloading of a given module, useful for when working interactively in the Python shell.
7. The three active namespaces that may exist during the execution of any given Python program are the _____, _____ and _____ namespaces.
ANSWERS: 1. True, 2. True, 3. Underscore, 4. False, 5. True, 6. reload, 7. built-in, global, local

COMPUTATIONAL PROBLEM SOLVING

7.4 Calendar Year Program (function version)

In this section, we implement and test the modular design for the calendar year program in section 7.2.

7.4.1 The Problem

The problem is the same as that given in Chapter 4—to display a calendar year from 1800 to 2099, displayed as shown in Figure 7-19.

7.4.2 Problem Analysis

Since the same problem is being solved, the algorithms utilized are also the same—an algorithm for computing the day of the week (for the years 1800 to 2099), an algorithm for storing the calendar year, and a means of displaying a calendar year three months across.

7.4.3 Program Design

The program requirements, data structures, algorithms, and overall program steps are the same as specified in Chapter 4. This program will only differ in that we will utilized a modular design for the program, as introduced in section 7.2.

This will result in a program that is more easily understood, more easily modified, and more amenable to program testing.

7.4.4 Program Implementation and Testing

Now that we are using modular design, we can more easily incrementally test the program. Given the specification of each module (function), we can first test each function separately to see if the implementation correctly meets the specification. The individual testing of modules is called **unit testing**. Once each module is individually tested, they can all be tested together.

```
2015

January                     February                    March
             1  2  3         1  2  3  4  5  6  7         1  2  3  4  5  6  7
 4  5  6  7  8  9 10         8  9 10 11 12 13 14         8  9 10 11 12 13 14
11 12 13 14 15 16 17        15 16 17 18 19 20 21        15 16 17 18 19 20 21
18 19 20 21 22 23 24        22 23 24 25 26 27 28        22 23 24 25 26 27 28
25 26 27 28 29 30 31                                    29 30 31

April                       May                         June
          1  2  3  4                     1  2             1  2  3  4  5  6
 5  6  7  8  9 10 11         3  4  5  6  7  8  9         7  8  9 10 11 12 13
12 13 14 15 16 17 18        10 11 12 13 14 15 16        14 15 16 17 18 19 20
19 20 21 22 23 24 25        17 18 19 20 21 22 23        21 22 23 24 25 26 27
26 27 28 29 30              24 25 26 27 28 29 30        28 29 30
                            31

July                        August                      September
          1  2  3  4                        1                1  2  3  4  5
 5  6  7  8  9 10 11         2  3  4  5  6  7  8         6  7  8  9 10 11 12
12 13 14 15 16 17 18         9 10 11 12 13 14 15        13 14 15 16 17 18 19
19 20 21 22 23 24 25        16 17 18 19 20 21 22        20 21 22 23 24 25 26
26 27 28 29 30 31           23 24 25 26 27 28 29        27 28 29 30
                            30 31

October                     November                    December
             1  2  3         1  2  3  4  5  6  7                1  2  3  4  5
 4  5  6  7  8  9 10         8  9 10 11 12 13 14         6  7  8  9 10 11 12
11 12 13 14 15 16 17        15 16 17 18 19 20 21        13 14 15 16 17 18 19
18 19 20 21 22 23 24        22 23 24 25 26 27 28        20 21 22 23 24 25 26
25 26 27 28 29 30 31        29 30                       27 28 29 30 31
```

FIGURE 7-19 Calendar Year Display

This is called **integration testing**. We demonstrate unit testing and integration testing of the calendar program in this section.

Implementation of the Calendar Year Program

Figure 7-20 shows an implementation of the Calendar Year program based on the modular design developed.

The main section of the program, **lines 188–207** , follows directly from our modular design, developed in section 7.2. The first function call is to function getYear (**lines 3–12**). This function simply contains an input statement and while loop to ensure that the year returned is in the range 1800 to 2099, inclusive.

Ne xt, function constructCalYear (**lines 121–147**) is called. As seen from the modular design, it relies on calls to functions leapYear, dayOfWeekJan1, numDaysInMonth, and constructCalMonth. This function essentially consists of a for loop that iterates once for each of the twelve months, calling constructCalMonth to construct each month and append each to the list calendar_year. The constructed year is a list of lists as given below,

[year, [month, week1, week2,...], [month, week1, week2,...],...]

```
1  # Calendar Year Program (Function Version)
2
3  def getYear():
4
5      """Returns a year between 1800-2099, inclusive, or the value -1."""
6
7      year = int(input('Enter year (yyyy) (-1 to quit): '))
8      while (year < 1800 or year > 2099) and year != -1:
9          print('INVALID INPUT - Year must be between 1800 and 2099')
10         year = int(input('Enter year (1-12): '))
11
12     return year
13
14 def leapYear(year):
15
16     """Returns True if year a leap year, otherwise returns False."""
17
18     if (year % 4 == 0) and (not (year % 100 == 0) or
19         (year % 400 == 0)):
20         leap_year = True
21     else:
22         leap_year = False
23
24     return leap_year
25
26 def dayOfWeekJan1(year, leap_year):
27
28     """Returns the day of the week for January 1 of a given year.
29
30         year must be between 1800 and 2099. leap_year must be True if
31         year a leap year, and False otherwise.
32     """
33     century_digits = year // 100
34     year_digits = year % 100
35     value = year_digits + (year_digits // 4)
36
37     if century_digits == 18:
38         value = value + 2
39     elif century_digits == 20:
40         value = value + 6
41
42     # adjust for leap years
43     if not leap_year:
44         value = value + 1
45
46     # return first day of month for Jan 1
47     return (value + 1) % 7
48
```

FIGURE 7-20 Implementation of the Calendar Year Program (*Continued*)

Functions dayOfWeekJan1 (**lines 26–47**) and numDaysInMonth (**lines 49–63**) are straightforward, neither relying on a call to any other function.

Function constructCalMonth is the largest of all the functions. It uses the same basic approach as before to properly align the weeks for display. In this function, a list of strings is created to be output to the screen. Finally, function

displayCalendar is given a particular calendar year structure and displays it in four rows of months, with three months across each row.

```
def numDaysInMonth(month_num, leap_year):
#-----------------------------------------------------------------------
    """Returns the number of days in a given month.

       month_num in the range 1-12, inclusive.
       leap_year True if month in a leap year, otherwise False.
    """
    num_days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

    # special check for February in leap year
    if (month_num == 2) and leap_year:
        num_days = 29
    else:
        num_days = num_days_in_month[month_num - 1]
    return num_days
```

```
def constructCalMonth(month_num, first_day, num_days_in_month):
#-----------------------------------------------------------------------
    """Returns a formatted calendar month for display on the screen.

       month_num in the range 1-12, inclusive.
       first_day in the range 0-6 (1-Sun, 2-Mon, ..., 0-Sat)

       Returns a list of strings of the form,
       [month_name, week1, week2, week3, week4, ...]
    """
    # init
    empty_str = ''
    blank_col = format(' ', '3')
    blank_week = format(' ', '21')
    month_names = ('January', 'February', 'March', 'April', 'May',
                   'June', 'July', 'August', 'September', 'October',
                   'November', 'December')

    calendar_month = [' ' + format(month_names[month_num - 1], '<20')]
    current_day = 1
    current_col = 1
    calendar_week = ''

    # init starting column
    if first_day == 0:
        starting_col = 7
    else:
        starting_col = first_day

    # add any needed leading spaces for first week of month
    while current_col < starting_col:
        calendar_week = calendar_week + blank_col
        current_col = current_col + 1

```

```
    # construct month for proper number of days
    while current_day <= num_days_in_month:

        # store day of month in field of length 3
        calendar_week = calendar_week + format(str(current_day), '>3')

        # append new week to month if at end of week
        if current_col == 7:
            calendar_month = calendar_month + [calendar_week]
            calendar_week = empty_str
            current_col = 1
        else:
            current_col = current_col + 1

        current_day = current_day + 1

    # if there is a final week, append to constructed month
    if calendar_week != empty_str:
        calendar_month = calendar_month + [calendar_week]

    return calendar_month

def constructCalYear(year):

    """Returns a formatted calendar year for display on the screen.

       year in the range 1800-2099, inclusive
       Returns a list beginning with the year, followed by
       twelve constructed months

          [year, month1, month2, week3, ..., month12]
    """
    # init
    leap_year = leapYear(year)
    first_day_of_month = dayOfWeekJan1(year, leap_year)
    calendar_year = [year]

    # construct calendar from twelve constructed months
    for month_num in range(1, 13):
        num_days_in_month = numDaysInMonth(month_num, leap_year)

        calendar_year = calendar_year + \
                      constructCalMonth(month_num, first_day_of_month,
                                        num_days_in_month)

        first_day_of_month = (first_day_of_month + \
                              num_days_in_month) % 7

    return calendar_year
```

```
149 def displayCalendar(calendar_year):
150
151     """Displays a calendar_year on the screen three months across. """
152
153     # init
154     month_separator = format(' ', '8')
155     blank_week = format(' ', '21')
156
157     # display year
158     print('\n', calendar_year[0])
159
160     # display months three across
161     for month_index in [1, 4, 7, 10]:
162
163         # init
164         week = 1
165         lines_to_print = True
166
167         while lines_to_print:
168
169             # init
170             lines_to_print = False
171
172             # print weeks of months side-by-side
173             for k in range(month_index, month_index + 3):
174                 if week < len(calendar_year[k]):
175                     print(calendar_year[k][week-1], end='')
176                     lines_to_print = True
177                 else:
178                     print(blank_week, end='')
179
180                 print(month_separator, end='')
181
182             # move to next screen line
183             print()
184
185             # increment week
186             week = week + 1
187
188 # --- main
189
190 # initialization
191 terminate = False
192
193 # program greeting
194 print('This program will display a calendar year for a given year')
195
```

FIGURE 7-20 Implementation of the Calendar Year Program (*Continued*)

Development and Unit Testing of Individual Modules

To begin, we can create a program that contains each of the function headers and docstring specifications, and implement and test them one by one. This is given in Figure 7-21, with an implementation for function getYear.

Running this version of the program will not produce any output. However, it

will define function getYear. Therefore, this function can be unit tested in the Python shell,
... getYear()
Enter year (yyyy) (21 to quit): 1800
1800

```
# continue to display calendar years until -1 entered
while not terminate:
    year = getYear()

    if year == -1:
        terminate = True
    else:
        # construct calendar
        calendar_year = constructCalYear(year)

        # display calendar
        displayCalendar(calendar_year)
```

FIGURE 7-20 Implementation of the Calendar Year Program

```
# Calendar Year Program
# This program will display any calendar year between 1800-2099.

def getYear():

    """Returns a year between 1800-2099, inclusive, or -1 if end of
       user input
    """

    year = int(input('Enter year (yyyy) (-1 to quit): '))

    while (year < 1800 or year > 2099) and year != -1:
        print('INVALID INPUT - Year must be between 1800 and 2099')
        year = int(input('Enter year: '))

    return year


#def leapYear(year):

#    """Returns True if year a leap year, otherwise returns False. """
.
.
```

FIGURE 7-21 Implementation and Unit Testing of Function getyear

... getYear()
Enter year (yyyy) (21 to quit): 2099
2099
... getYear()

Enter year (yyyy) (21 to quit): 1985
1985
... getYear()
Enter year (yyyy) (21 to quit): 1799
INVALID INPUT – Year must be between 1800 and 2099 Enter year: 2100
INVALID INPUT – Year must be between 1800 and 2099 Enter year: 2050
2050
...

Since the getYear function tested OK, we can next implement and test the leapYear function. This is given in Figure 7-22.

```
# Calendar Year Program
# This program will display any calendar year between 1800-2099.

##def getYear():

##    """Returns a year between 1800-2099, inclusive, or -1 if end of
##       user input.
##    """
##    year = int(input('Enter year (yyyy) (-1 to quit): '))
##
##    while (year < 1800 or year > 2099) and year != -1:
##        print('INVALID INPUT - Year must be between 1800 and 2099')
##        year = int(input('Enter year: '))
##
##    return year

def leapYear(year):

    """Returns True if year a leap year, otherwise returns False."""

    if (year % 4 == 0) and (not (year % 100 == 0) or \
        (year % 400 == 0)):
        leap_year = True
    else:
        leap_year = False

    return leap_year
.
.
```

FIGURE 7-22 Implementation and Unit Testing of Function leapYear

Running this version of the program defines function leapYear, which can then be unit tested.

... leapYear(1803)
False
... leapYear(1800)

```
False
... leapYear(1860)
True
... leapYear(1900)
False
... leapYear(1964)
True
... leapYear(2000)
True
... leapYear(2012)
True
```

Based on the test results, both getYear and leapYear are properly working. Next, we will unit test function dayOfWeekJan1. The implementation of this function is given in Figure 7-23.

```
.
.

##def leapYear(year):

##    """Returns True if year a leap year, otherwise returns False. """
##
##    if (year % 4 == 0) and (not (year % 100 == 0) or
##        (year % 400 == 0)):
##        leap_year = True
##    else:
##        leap_year = False
##
##    return leap_year

def dayOfWeekJan1(year, leap_year):

    """Returns the day of the week for January 1 of a given year.

        year must be between 1800 and 2099. leap_year must be True if
        year a leap year, and False otherwise.
    """
    century_digits = year // 100
    year_digits = year % 100
    value = year_digits + (year_digits // 4)

    if century_digits == 18:
        value = value + 2
    elif century_digits == 20:
        value = value + 6

    # adjust for leap years
    if not leap_year:
        value = value + 1

    # return first day of month for Jan 1
    return (value + 1) % 7

.
.
```

FIGURE 7-23 Implementation and Unit Testing of Function dayOfWeekJan1

As with previous stages of development, running this version of the program defines function leapYear, which we can then unit test.

... dayOfWeekJan1(1800, False)
4
... dayOfWeekJan1(1864, True)
6
... dayOfWeekJan1(1900, False)

2
... dayOfWeekJan1(2000, True)
0
... dayOfWeekJan1(2012, True)
1
... dayOfWeekJan1(2013, False)
3

Recalling that the values 0–6 represent the days of the week Saturday to Sunday (where Saturday is 0), checking these results with other sources, they are found to be correct. Next, we unit test function numDaysInMonth as shown in Figure 7-24.

```
.
.

##    # adjust for leap years
##    if not leap_year:
##        value = value + 1
##
##    # return first day of month for Jan 1
##    return (value + 1) % 7


def numDaysInMonth(month_num, leap_year):

    """Returns the number of days in a given month.

        month_num in the range 1-12, inclusive.
        leap_year True if month in a leap year, otherwise False.
    """
    num_days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

    # special check for February in leap year
    if (month_num == 2) and leap_year:
        num_days = 29
    else:
        num_days = num_days_in_month[month_num]

    return num_days
```

FIGURE 7-24 Implementation and Unit Testing of Function numDaysInMonth

By executing this version of the program and unit testing function numDaysInMonth, we get the following results.

... numDaysInMonth(1, False)
28

```
... numDaysInMonth(2, False)
31
... numDaysInMonth(2, True)
29
... numDaysInMonth(3, False)
30
... numDaysInMonth(12,False)
Traceback (most recent call last):

File “ pyshell#5 .”, line 1, in ,module .
numDaysInMonth(12,False)
File “C:\My Python Programs\CalendarYearFunc.py”, line 65, in
numDaysInMonth
num_days num_days_in_month[month_num]
IndexError: tuple index out of range
...
```

Obviously, something is wrong with this function. First, we have to make sure that we called it with proper values. The function specification indicates that it should be given values in the range 1 to 12, inclusive, as the first argument, and a Boolean value (indicating whether the year is a leap year or not) for the second. Therefore, it is being called correctly.

Looking at the results produced for the consecutive months tested, it seems that the results are off by one month. That is, numDaysInMonth(1, False) is giving the result 28, which would be correct for February (of a non-leap year), and numDaysInMonth(2, False) is giving the result 31, which would be the correct results for March (of either a leap year or a non-leap year). So, if the index value used to index list num_days_in_month were one greater than it should be, this would be consistent with the test results. Furthermore, the fact that an index out of range error occurs for a month number of 12 further supports the likely “off by one” error.

However, there is one result that is inconsistent with this hypotheses. For February of a leap year, numDaysInMonth(2, True), we get the correct result of 29. This could be explained if the case of February in a leap year is handled as a special case in the function code. In fact, that is exactly the case by the if-else statement. Therefore, we make the needed correction in the function, given in Figure 7-25.

```
def numDaysInMonth(month_num, leap_year):

    """Returns the number of days in a given month.

        month_num in the range 1-12, inclusive.
        leap_year True if month in a leap year, otherwise False.
    """
    num_days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

    # special check for February in leap year
    if (month_num == 2) and leap_year:
        num_days = 29
    else:
        num_days = num_days_in_month[month_num - 1]

    return num_days
```

FIGURE 7-25 Corrected numDaysInMonth Function

We again test the function, and this time get the correct result for each test case.

... numDaysInMonth(1, False)
31
... numDaysInMonth(2, False)
28
... numDaysInMonth(2, True)
29
... numDaysInMonth(3, False)
31
... numDaysInMonth(4, False)
30
etc.

Since there are only 13 possible sets of input values to the function (including two sets for February), we test each one and find that the function is working for each of these cases.

Function constructCalMonth is tested next. It is passed a month number (1 to 12), a day of the week value (0 to 6) and the number of days in the month, and returns a constructed calendar month as given below,

... constructCalMonth(1, 0, 31)
[' January ', ' 1', ' 2 3 4 5 6 7 8', ' 9 10 11 12 13 14 15', ' 16 17 18 19 20 21 22', ' 23 24 25 26 27 28 29', ' 30 31']
...

Additional testing of this function indicates that it is working correctly.
Integration Testing of Modules

The remaining functions to test are constructCalYear and displayCalendar. Function constructCalYear relies on the use of functions leapYear, dayOfWeekJan1, numDaysInMonth, and constructCalMonth. Since each of these has been tested and found correct, we can now perform integration testing of all these functions with function constructCalYear. We therefore execute a version of the program in which all the code developed so far, except these functions, are commented out, as depicted in Figure 7-26.

```
##
##    # if there is a final week, append to constructed month
##    if calendar_week != empty_str:
##        calendar_month = calendar_month + [calendar_week]
##
##    return calendar_month

def constructCalYear(year):

    """Returns a formatted calendar year for display on the screen.

        year in the range 1800-2099, inclusive
        Returns a list beginning with the year, followed by
        twelve constructed months

          [year, month1, month2, week3, ..., month12]
    """
    # init
    leap_year = leapYear(year)
    first_day_of_month = dayOfWeekJan1(year, leap_year)
    calendar_year = [year]

    # construct calendar from twelve constructed months
    for month_num in range(1, 13):
        num_days_in_month = numDaysInMonth(month_num, leap_year)

        calendar_year = calendar_year + \
                        constructCalMonth(month_num, first_day_of_month,
                                          num_days_in_month)

        first_day_of_month = (first_day_of_month + num_days_in_month) % 7

    return calendar_year
```

FIGURE 7-26 Integration Testing with Function constructCalYear

Test results from a call to ConstructCalYear are given below.
... constructCalYear(2012)

[2012, [' January ', ' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13 14 ', ' 15 16 17 18 19 20 21', ' 22 23 24 25 26 27 28', ' 29 30 31'], [' February

', ' 1 2 3 4', ' 5 6 7 8 9 10 11', ' 12 13 14 15 16
17 18', ' 19 20 21 22 23 24 25', ' 26 27 28 29'], [' March ', ' 1 2 3', ' 4 5 6 7 8 9
10', ' 11 12 13 14 15 16 17', ' 18 19 2
0 21 22 23 24', ' 25 26 27 28 29 30 31', [' April ', ' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13
14', ' 15 16 17 18 19 20 21', ' 22 23 24 25 26
27 28', ' 29 30'], [' May ', ' 1 2 3 4 5', ' 6 7 8
9 10 11 12', ' 13 14 15 16 17 18 19', ' 20 21 22 23 24 25 26', ' 27 28 29 30 3
1'], [' June ', ' 1 2', ' 3 4 5 6 7 8 9', ' 10 11 12 13 14 15 16', ' 17 18 19 20 21 22 23',
' 24 25 26 27 28 29 30'], [' July ', ' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13 14', ' 15 16
17 18 19 20 21', ' 22 23 24 25 26 27 28', ' 29 30 31'], [' August ', ' 1 2 3 4', ' 5 6 7
8 9 10 11', ' 12 13 14 15 16 17 18', ' 19 20 21 22 23 24 25', ' 26 27 28 29 30 31'],
[' September ', ' 1', ' 2 3 4 5 6 7 8', ' 9 10 11 12 13 14 15', ' 16 17 18 19
20 21 22', ' 23 24 25 26 27 28 29', ' 30'], [' October ', ' 1 2 3 4 5 6', ' 7 8 9 10 11
12 13', ' 14 15 16 17 18 19 20', ' 21 22 23 24
25 26 27', ' 28 29 30 31'], [' November ', ' 1 2 3', ' 4 5 6 7 8 9 10', ' 11 12 13 14 15
16 17', ' 18 19 20 21 22 23 24', ' 25 2
6 27 28 29 30',] [' December ', ' 1', ' 2 3 4 5 6 7 8', ' 9 10 11 12 13 14 15', ' 16 17
18 19 20 21 22', ' 23 24 25 26 27
28 29', ' 30 31']]
...

The constructed year looks correct, and so we include the final function to be tested, displayCalendar, in this integration testing. Test results from a call to displayCalendar are given below.

This program will display a calendar year for a given year Enter year (yyyy) (21 to quit): 2012
2012
J
a 1 8
n
u
a 2 9
r
y 1

3 0
1
4 1
1
5 2
.
.
.

This output does not look even close to what it should be. The output displayed by function displayCalendar depends on providing it with a correctly structured calendar year. So maybe we should think through each step of function displayCalendar with the test output from function constructCalYear to see if we can gain any insight into the problem. The output from function constructCalYear for the year 2012 is partially reproduced below.

... constructCalYear(2012)
[2012, ' January ', ' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13 14 ', ' 15 16 17 18 19 20 21', ' 22 23 24 25 26 27 28', ' 29 30 31'], [' February

', ' 1 2 3 4', ' 5 6 7 8 9 10 11', ' 12 13 14 15 16 17 18', ' 19 20 21 22 23 24 25', ' 26 27 28 29'], [' March ', '
The body of function constructCalYear, for parameter year, is shown below.

1 # init
2 leap_year 5 leapYear(year)
3 first_day_of_month 5 dayOfWeekJan1(year, leap_year)
4 calendar_year 5 [year]
5
6 # construct calendar from twelve constructed months
7 for month_num in range(1, 13):
8 num_days_in_month 5 numDaysInMonth(month_num, leap_year)
9
10 calendar_year 5 calendar_year 1 \
11 constructCalMonth(month_num, first_day_of_month,
12 num_days_in_month)
13
14 first_day_of_month 5 (first_day_of_month 1 num_days_in_month) % 7
15

16 return calendar_year

Lines 2 and **3** make calls to functions leapYear and dayOfWeekJan1, which have both been tested, so we can assume, for now, that the results of these function calls are correct. On **line 4**, list calendar_year is initialized to a list of one element, the integer representing the year. The for loop at **line 7** is used to construct each of the calendar months one at a time, appending each to list calendar_year.

One thing to double-check is the range of the for loop, since there is always a chance that we have an “off by one” error. In this case, range is called with a first argument of 1, and a second argument of 13. Thus, variable month_num will be assigned to the values 1,2,...,12. The loop will iterate, therefore, 12 times—once for each of the 12 months. This is the correct number of iterations. Now we should check to see if they are the correct range of values.

V ariable month_num is only used in the call to function constructCalMonth on **line 11**. Thus, we check the specification of the function to see if it requests month values in the range 1–12, or if it requests their index values 0–11. Looking back at the specification for this function in Figure 7-8, 1–12 is the correct range of month values to be passed to it.

Next, we consider the concatenation of calendar_year with a newly constructed calendar month each time through the loop. This occurs in **lines 10–12**, given below.
calendar_year 5 calendar_year 1 \
constructCalMonth(month_num,first_day_of_month, num_days_in_month) We know that calendar_year is initialized to the list [2012]. Also, calls to constructCalMonth return a constructed calendar month as a list of the form,
[' January ', ' 1', ' 2 3 4 5 6 7 8', ' 9 10 11 12 13 14 15', ' 16 17 18 19 20 21 22', ' 23 24 25 26 27 28 29', ' 30 31']
Thus, the above instruction would result in the following,

calendar_year 5 [2012] 1 [' January ', ' 1', ' 2 3 4 5 6 7 8', ' 9 10 11 12 13 14 15', ' 16 17 18 19 20 21 22', ' 23 24 25 26 27 28 29', ' 30 31']

Now we realize something. The structure of a calendar year is designed to be,
[year, [month, week1, week2,...], [month, week1, week2,...],....]
So we check the initial result of the structure of calendar_year after doing a

simplified version of the above concatenation in the Python shell,
... [2012] 1 [' January ', ' 1'] [2012, ' January ', ' 1']

Here is a problem! The result of the concatenations should be a list of lists, with each sublist containing a constructed month. The result above is a *single* list of values. That would certainly be a cause for the displayCalYear function not working correctly. The error is that instead of concatenating the two lists, each constructed month should be *appended* to the calendar year list, as given below.

```
calendar_year.append(constructCalMonth(
month_num,
first_day_of_month,
num_days_in_month))
```

Testing again in the Python shell, this gives the following (correct) results.
... [2012] 1 [[' January ', ' 1']] [2012, [' January ', ' 1']]

Therefore, we make the correction and retest function displayCalYear. This time we get a correctly displayed calendar year, as shown in Figure 7-27. (Note that the width of the shell window needs to be adjusted for the weeks of the calendar year to properly align.)

```
This program will display a calendar for a given year
Enter year (yyyy) (-1 to quit): 2012

 2012
 January                        February                       March
  1  2  3  4  5  6  7                  1  2  3  4                        1  2  3
  8  9 10 11 12 13 14           5  6  7  8  9 10 11           4  5  6  7  8  9 10
 15 16 17 18 19 20 21          12 13 14 15 16 17 18          11 12 13 14 15 16 17
 22 23 24 25 26 27 28          19 20 21 22 23 24 25          18 19 20 21 22 23 24

 April                          May                            June
  1  2  3  4  5  6  7                 1  2  3  4  5                          1  2
  8  9 10 11 12 13 14           6  7  8  9 10 11 12           3  4  5  6  7  8  9
 15 16 17 18 19 20 21          13 14 15 16 17 18 19          10 11 12 13 14 15 16
 22 23 24 25 26 27 28          20 21 22 23 24 25 26          17 18 19 20 21 22 23

 July                           August                         September
  1  2  3  4  5  6  7                  1  2  3  4                             1
  8  9 10 11 12 13 14           5  6  7  8  9 10 11           2  3  4  5  6  7  8
 15 16 17 18 19 20 21          12 13 14 15 16 17 18           9 10 11 12 13 14 15
 22 23 24 25 26 27 28          19 20 21 22 23 24 25          16 17 18 19 20 21 22
                                                             23 24 25 26 27 28 29

 October                        November                       December
     1  2  3  4  5  6                     1  2  3                             1
  7  8  9 10 11 12 13           4  5  6  7  8  9 10           2  3  4  5  6  7  8
 14 15 16 17 18 19 20          11 12 13 14 15 16 17           9 10 11 12 13 14 15
 21 22 23 24 25 26 27          18 19 20 21 22 23 24          16 17 18 19 20 21 22
                                                             23 24 25 26 27 28 29

Enter year (yyyy) (-1 to quit):
```

FIGURE 7-27 Calendar Year Program Output

CHAPTER SUMMARY General Topics Python-Specific Programming Topics

Modules/Module Interface/Specification Top-Down Design
Stacks
Namespaces and Name Clashes
Global Scope/Global Variables
Python Docstring Specification
Built-in/Global/Local Namespaces in Python Main Modules/Standard Modules in Python Forms of Module Import in Python
Global vs. Private Module Variables in Python Module Loading and Execution in Python

CHAPTER EXERCISES Section 7.1

1. For the following function, def hours_of_daylight(month, year)

(a) Give an appropriate docstring specification where hours_of_daylight returns the total number of hours of daylight for the month and year given (each passed an integer value) designed so that the function does not check for invalid parameter values.

(b) Give a print statement that displays the docstring for this function.

Section 7.2

2. Develop a modular design depicting the components of a typical computer, similar to that for the Space Shuttle in Figure 7-2. Assume that the major components of a computer system consist of a CPU (central processing unit), busses (connections between components), main memory (RAM), secondary memory (hard drive, USB drive, etc.), and input/output devices (mouse, keyboard, etc.). Search online for more specific subsystems and devices to complete your design.

3. For the hours_of_daylight function in exercise 1, give a code segment that prompts the user for a month and year, and appropriately calls function hours_of_daylight according to its docstring specification, displaying the result.

Section 7.3

4. For module1, module2, and the client module shown below, indicate which of the imported identifiers would result in a name clash if the imported identifiers were not fully qualified.

```
# module1

def func_1(n):
    etc.

def func_2(n):
    etc.
```

```
# module2

def func_2(n):
    etc.

def func_3(n):
    etc.
```

```
from module1 import *
from module2 import *

def func_3(n):
    etc.
```

5. Depict what is left on stack s after the following series of push and pop operations (starting with the first column of operations and continuing with the second). Assume that the stack is initially empty.

push(s,10) push(s,50)
push(s,20) push(s,60)
push(s,40) push(s,80)
pop(s) pop(s)
pop(s) pop(s)

6. For the program in Figure 7-9 that imports modules module1 and module1, indicate how many total namespaces exist for this program.
7. For the Palindrome Checker program in section 7.3.7, describe the changes that would be needed in the program if the import statement were changed from import Stack to from Stack import *.
8. For the following program and the imported modules, describe any name clashes that would occur for both program version1 and version 2.

```
# module m1

def total(items):

def convert(items):

def show(items)
```

```
# module m2

def totalSum(items):

def convert(items):

def display(items)
```

```
from m1 import *
from m2 import *

def display()

def calc()

def getItems()

# ---- main

items = getItems()
items = convert(items)
show(items)
```

Version 1

```
import m1
import m2

def display()

def calc()

def getItems()

# ---- main

items = getItems()
items = m2.convert(items)
display(items)
```

Version 2

PYTHON PROGRAMMING EXERCISES

P1. Write a function called convertStatus that is passed status code 'f', 's', 'j', or 'r' and returns the string 'freshman', 'sophomore', 'junior', or 'senior', respectively. Design your function so that if an inappropriate letter is passed, an error value is returned. Make sure to include an appropriate docstring with your function.

P2. Write a function called palindromeChecker using iteration to return True if a provided string is a palindrome, and False otherwise. Make sure to include

docstring specification for the function.
P3. Implement a set of functions called getData, extractValues, and calcRatios. Function getData should prompt the user to enter pairs of integers, two per line, with each pair read as a single string, for example,

Enter integer pair (hit Enter to quit):
134 289 (read as '134 289')
etc.

These strings should be passed one at a time as they are being read to function extractValue, which is designed to return the string as a tuple of two integer values,
extractValues('134 289') returns (134, 289) etc.
Finally, each of these tuples is passed to function calcRatios one at a time to calculate and return the ratio of the two values. For example,
calcRatios((134, 289)) returns 0.46366782006920415 etc.
Implement a complete program that displays a list of ratios for an entered series of integer value pairs. Make sure to include docstring specification for each of the functions.

PROGRAM MODIFICATION PROBLEMS

M1. Stack Module: Limited Stack Size
For the stack module given in Figure 7-15, suppose that the implementation is to limit stacks to no more than 100 items. Redesign and re-implement the relevant parts of the module that need to be changed. Write a small program to demonstrate this new version of the stack module.

M2. Stack Module: Ability to “Peek” in Stack
For the stack module given in Figure 7-15, redesign and re-implement the relevant parts of the module to allow the ability to “peek” into the stack to find out if a given element is on the stack or not. Write a small program to demonstrate this new version of the stack module.

M3. Stack Module: Double-Ended Stacks
A double-ended stack is essentially a pair of stacks that share a fixed amount of storage (memory). The two stacks are designed so that the top of stack of each begins at each end of a list structure. Each top of stack moves towards the other when an element is pushed, as depicted below. This stack implementation has

the advantage of more effectively utilizing a fixed amount of memory than if each stack were allocated its own storage.

Modify and test the stack module to behave as a double-ended stack.

M4. Calendar Year Program: Optional Month/Year Display

Modify the Calendar Year Program so the user can select whether they want to display a complete calendar year, or just a specific calendar month.

M5. Calendar Year Program: Flexible Layout of Months

Modify the Calendar Year Program so the user can select whether they want the calendar displayed "row
oriented" (with January, February, and March in the first row, April, May, and June in the second row,
etc.) or "column oriented" (with January, February, March, and April in the first column, May, June, July,
and August in the second column, etc.).

PROGRAM DEVELOPMENT PROBLEMS

D1. Parentheses Matching Program

Implement and test a Python program that determines if all parentheses in an entered line of code form matching pairs. Note: Pairs of parentheses may be nested.

D2. Determining Body Mass Index (BMI)

Following is a chart for determining one's body mass index (BMI). BMI is a general indication of the amount of body fat that a person has. The formula for computing BMI is,

BMI 5 mass / height^2

Implement a Python program that prompts a user for their height and weight. Height should be entered as inches and weight should be entered in pounds. Perform the calculation in units of kilograms and meters as shown in the chart.

Compare the result to the information in the chart. Use functions in your program design.

D3. Create interesting geometric animations by the use of turtle graphics and the stack module provided in the chapter. A snapshot of an example is shown below.

To do this, create a geometric shape in turtle graphics. Then, create multiple invisible turtle objects of this shape, each scaled down to a smaller size than the previous and pushed on a stack. Once the stack has been “loaded” with a sufficient number of such turtle objects, continually pop the stack and make each visible on the screen, pushing the popped turtle object onto a second stack. When the first stack is empty, reverse the process using the second stack (pushing the popped turtle objects from the second stack back onto the first stack), making each visible again. Use the sleep function of the time module of Python to control the speed of the animation.

Text Files CHAPTER 8

We have, up to now, been storing data only in the variables and data structures of programs. However, such data is not available once a program terminates. Therefore, in order for information to persist from one program execution to the next, the data must be stored in a data file. In this chapter we discuss the use of one particular type of data file, text files.

OBJECTIVES

After reading this chapter and completing the exercises, you will be able to: ♦ Explain what a text file is

♦ Differentiate between a text file and a binary file
♦ Explain the process of opening and closing a file
♦ Explain the process of reading and writing files
♦ Explain the process of exception handling
♦ Explain the process of unit testing using test drivers ♦ Effectively utilize text files in Python
♦ Effectively perform string processing in Python
♦ Catch and handle exceptions in Python

CHAPTER CONTENTS

Motivation
Fundamental Concepts
8.1 What Is a Text File?
8.2 Using Text Files
8.3 String Processing
8.4 Exception Handling
Computational Problem Solving
8.5 Cigarette Use/Lung Cancer Correlation Program

MOTIVATION

The vast amount of information being generated today means that there must also be vast amounts of storage for all this data. Information sharing on the web through Wikipedia, social networks, and other means keeps driving this growth. *It has been estimated that 90% of all the data in the world has been generated in the last two years.*

Throughout the age of computing, data has always been shared in some manner. In the early days data had to be physically carried on storage media such as reels of magnetic tapes. Later, early developments in computer networks allowed the sharing of data between a small number of large and expensive connected computers. With the advent of the Internet, computers are not directly connected, but indirectly through routers that temporarily store and forward data toward its destination. The World Wide Web has made access to information easy and intuitive by the incorporation of hypertext— text that can be clicked on to retrieve more text—effectively making text “three-dimensional.”

The evolving technology of “cloud computing” is a further step in the sharing of information. Not only is data easily shared, but also the programs and other services needed to use that data. Figure 8-1 shows the various information storage devices in use today. (Note that a volatile memory device is one that loses its contents when power is lost, whereas a nonvolatile device retains its contents.)

Storage Technology	Example	Characteristics
Magnetic Storage	Hard drive	Nonvolatile
	Magnetic Tape Storage	Nonvolatile
Semiconductor Memory	Main memory	Volatile
	USB (Thumb) Drive	Nonvolatile
Optical Storage	CD, DVD	Nonvolatile

FIGURE 8-1 Types of Storage Technology

FUNDAMENTAL CONCEPTS

8.1 What Is a Text File?

A **text file** is a file containing characters, structured as individual lines of text. In addition to printable characters, text files also contain the *nonprinting* newline character, \n, to denote the end of each text line. As discussed in Chapter 2, the newline character causes the screen cursor to move to the beginning of the next screen line. Thus, text files can be directly viewed and created using a text editor.

In contrast, **binary files** can contain various types of data, such as numerical values, and are therefore not structured as lines of text. Such files can only be read and written via a computer program. Any attempt to directly view a binary file will result in “garbled” characters on the screen. Our purpose is not to cover all of types of files in Python. Rather, we cover enough to be able to perform simple reading and writing of text files.

LET’S TRY IT Let’s view both a text file and a binary file using a simple text editor like notepad. First, create a simple file within IDLE named hello.py containing only two lines :

```
print 'Hello'
print 'There'
```

Execute the program. From the shell window that the program displays the results in, enter the following, ... import hello

This will both execute the program and compile it into a binary file named hello.pyc. Open the Python source file using notepad (or other simple text editor). The two print statements of the program should be displayed. Open the Python compiled file of this program using notepad and observe what is displayed.

A **text file** is a file containing characters, structured as lines of text. A **binary file** is a file that is formatted in a way that only a computer program can read.

8.2 Using Text Files

Fundamental operations of all types of files include *opening* a file, *reading* from a file, *writing* to a file, and *closing* a file. Next we discuss each of these operations when using text files in Python.

8.2.1 Opening Text Files

All files must first be opened before they can be read from or written to. In

Python, when a file is (successfully) opened, a file object is created that provides methods for accessing the file. We look how to open files for either reading (from) or writing (to) a file in this section.

All files must first be opened before they can be used. In Python, when a file is opened, a file object is created that provides methods for accessing the file.
Opening for Reading
To open a file for reading, the built-in open function is used as shown,
input_file 5 open('myfile.txt','r')

The first argument is the file name to be opened, 'myfile.txt'. The second argument, 'r', indicates that the file is to be opened for reading. (The second argument is optional when opening a file for reading.) If the file is successfully opened, a file object is created and assigned to the provided identifier, in this case identifier input_file.

When opening a file for reading, there are a few reasons why an *I/O error* may occur. (We look at how to include the ability of a program to catch and handle such errors in the discussion of exception handling in section 8.4.) First, if the file name does not exist, then the program will terminate with a “no such file or directory” error,

... open('testfile.txt','r')
Traceback (most recent call last):

File " pyshell#1 .", line 1, in ,module .
open('testfile.txt','r')
IOError: [Errno 2] No such file or directory: 'testfile.txt'

This error can also occur if the file name is not found in the location looked for (uppercase and lowercase letters are treated the same for file names). When a file is opened, it is first searched for in the same folder/directory that the program resides in. The programs in the text are written this way. However, an alternate location can be specified in the call to open by providing a path to the file,

input_file 5 open('data/myfile.txt','r')

In this case, the file is searched for in a subdirectory called data of the directory in which the program is contained. Thus, its location is relative to the program location. (Although some operating systems use forward slashes, and other

backward slashes in path names, directory paths in Python are always written with forward slashes and are automatically converted to backward slashes when required by the operating system executing on.) Absolute paths can also be provided giving the location of a file anywhere in the file system,

input_file 5 open('C:/mypythonfiles/data/myfile.txt','r') When the program has finished reading the file, it should be closed by calling the close method on the file object,
input_file.close() Once closed, the file may be reopened (with reading starting at the beginning of the file) by the same, or another program. Next, we look at how to open files for writing in Python.
LET'S TRY IT In a new Python file window, enter the following lines,

Line one
Line two
Line three

Save the file under the name data.txt. (Make sure to save it with extension '.txt' and not '.py'.) Then, in the same folder (directory) as the data file, create the following Python program, file_name 5 input('Enter file name: ')
input_file 5 open(file_name, 'r')
Run this program twice. The first time, enter the file name garbage.txt. The second time, enter the correct file name data.txt and observe the results.
To open a file for reading in Python, the built-in function open is called with (optional) argument value 'r'.
Opening for Writing
To open a file for writing, the open function is used as shown below,
output_file 5 open('mynewfile.txt','w')

Note that, in this case, 'w' is used to indicate that the file is to be opened for writing. If the file already exists, it will be overwritten (starting with the first line of the file). When using a second argument of 'a', the output will be appended to an existing file instead.

It is important to close a file that is written to, otherwise the tail end of the file may not be written to the file (discussed below),
output_file.close()

When opening files for writing, there is not much chance of an I/O error

occurring. The provided file name does not need to exist since it is being created (or overwritten). Thus, the only error that may occur is if the file system (such as the hard disk) is full.

LET’S TRY IT In a new Python file window, enter the following lines,

file_name 5 input('Enter file name: ')
file 5 open(file_name, 'w')
file.close()

Save the file under the name createfile.py and run it. When the program requests a file name, give it any file name you wish with the extension '.txt'. Then, look in the folder in which the program resides to see if a new file with the file name that you entered exists. (This file will be empty—we will see how to write to a file next.)

Modify the open instruction above, changing 'w' to 'r' and rerun the program. When it requests a file name, enter the name of the file you just created. Run it a second time; this time give it the wrong file name. Observe the different results in each case.

To open a file for writing in Python, the built-in function open is called with a second argument of 'w'. A second argument of 'a' will open a file for appending to instead.

8.2.2 Reading Text Files

The readline method returns as a string the next line of a text file, including the end-of-line character, \n. When the end-of-file is reached, it returns an empty string as demonstrated in the while loop of Figure 8-2.

Text File myfile.txt

```
Line One\n
Line Two\n
Line Three\n
```

```
input_file = \
   open('myfile.txt','r')
empty_str = ''

line = input_file.readline()

while line != empty_str:
    print(line)
    line = input_file.readline()

input_file.close()
```

Screen Output

```
Line One

Line Two

Line Three
```

FIGURE 8-2 Reading from a Text File

It is also possible to read the lines of a file by use of the for statement,

input_file 5 \
open('myfile.txt','r')
for line in input_file:

Using a for statement, *all* lines of the file will be read one by one. Using a while loop, however, lines can be read until a given value is found, for example.

Finally, note the blank lines in the screen output. Since read_line returns the newline character, and print adds a newline character, *two* newline characters are output for each line displayed, resulting in a skipped line after each. We will see an easy way to correct this when we discuss string methods.

LET'S TRY IT Create a text file named 'myfile.txt'. Enter and execute the following program and observe the results. Make sure that the program and text file reside in the same folder.
input_file 5 open('myfile.txt','r')
for line in input_file:
print(line)
The readline method returns the next line of a text file, including the end-of-line character. If at the end of the file, an empty string is returned.
8.2.3 Writing Text Files
The write method is used to write strings to a file, as demonstrated in Figure 8-3.

Text File
myfile.txt

```
Line One\n
Line Two\n
Line Three\n
```

```
empty_str = ''
input_file = open('myfile.txt','r')
output_file = open('myfile_copy.txt','w')

line = input_file.readline()

while line != empty_str:
    output_file.write(line)
    line = input_file.readline()

output_file.close()
```

Text File
myfile_copy.txt

```
line one
line two
line three
```

FIGURE 8-3 Writing to a Text File

This code copies the contents of the input file, 'myfile.txt', line by line to the output file, 'myfile_ copy.txt'. In contrast to print when writing to the screen, *the write method does not add a newline character to the output string*. Thus, a newline character will be output only if it is part of the string being written. In this case, each line read contains a newline character.

Finally, when writing to a file, data is first placed in an area of memory called a *buffer*. Only when the buffer becomes full is the data actually written to the file.

(This makes reading and writing files more efficient.) Since the last lines written may not completely fill the buffer, the last buffer's worth of data may not be written. The close() method *flushes* the buffer to force the buffer to be written to the file.

LET'S TRY IT In the Python shell, open an existing file myfile.txt and do the following.

```
... input_file 5 ('myfile.txt','r')
... output_file 5 ('newfile.txt','w')
... line 5 input_file.readline()
... output_file.write(line)
... output_file.close()
```

Observe that newfile.txt has been created, and examine its contents.
Use the write()method to output text to a file. To ensure that all data has been written, call the close() method to close the file after all information has been written.

Self-T est Questions
1. Only files that are written to need to be opened first. (TRUE/FALSE)

2. Indicate which of the following reasons an IOError (exception) may occur when opening a file. **(a)** Misspelled file name **(c)** File not found in directory searched **(b)** Unmatched uppercase and lowercase letters

3. Which one of the following is true?

(a) When calling the built-in open function, a second argument of 'r' or 'w' must always be given
(b) When calling the built-in open function, a second argument of 'r' must always be given when opening a file for reading
(c) When calling the built-in open function, a second argument of 'w' must always be given when opening a file for writing

4. Which one of the following is true?
(a) There is more chance of an I/O error when opening a file for reading.
(b) There is more chance of an I/O error when opening a file for writing.

5. The readline method reads every character from a text file up to and including

the next newline character '\n'. (TRUE/FALSE)
6. It is especially important to close a file that is open for writing. (TRUE/FALSE)
ANSWERS: 1. False, 2. (a),(c), 3. (c), 4. (a), 5. True, 6. True

8.3 String Processing

The information in a text file, as with all information, is most likely going to be searched, analyzed, and/or updated. Collectively, the operations performed on strings is called *string processing*. We have already seen some operations on strings—for example, str[k], for accessing individual characters, and len(str)for getting the length of a string. In this section, we revisit sequence operations that apply to strings, and look at additional stringspecific methods.

String processing refers to the operations performed on strings that allow them to be accessed, analyzed, and updated.

8.3.1 String Traversal

We saw in Chapter 4 how any sequence can be traversed, including strings. This is usually done by the use of a for loop. For example, if we want to read a line of a text file and determine the number of blank characters it contains, we could do the following,

space 5 ' '
num_spaces 5 0

line 5 input_file.readline()
for k in range(0,len(line)): if line[k] 55 space:

num_spaces 5 num_spaces 1 1
We also saw that the last lines can be done more simply without the explicit use of an index variable,
for chr in line:
if chr 55 space:
num_spaces 5 num_spaces 1 1 Given the ability to traverse a string, each character can be individually “looked at” for various types of string processing. We look at some string processing operations next.
The characters in a string can be easily traversed, without the use of an explicit index variable, using the for *chr* in *string* form of the for statement.

8.3.2 String-Applicable Sequence Operations

Because strings (unlike lists) are immutable, sequence-modifying operations are not applicable to strings. For example, one cannot add, delete, or replace characters of a string. Therefore, *all string operations that "modify" a string return a new string that is a modified version of the original string*. Sequence operations relevant to string processing are given in Figure 8-4. We look at stringspecific operations in the next section.

Sequences Operations Applicable to Strings			
Length	`len(str)`	Membership	`'h' in s`
Select	`s[index_val]`	Concatenation	`s + w`
Slice	`s[start:end]`	Minimum Value	`min(s)`
Count	`s.count(char)`	Maximum Value	`max(s)`
Index	`s.index(char)`	Comparison	`s == w`

FIGURE 8-4 Sequence Operations on Strings

Recall that as we saw with lists, the slice operator s[start:end] returns the substring starting with index start, *up to but not including* index end. Also, s.index(chr) returns the index of the first occurrence of chr in s. Finally, min and max as applied to strings return the smallest (largest) character based on the underlying Unicode encoding. Thus, for example, all lowercase letters are larger (have a larger Unicode value) than all uppercase letters. We give examples of each of these operations for s 5'Hello Goodbye!'.

... len(s) s.count('o') ... s 1 '!!' 14 3 'Hello Goodbye!!!' ... s[6] ... s.index('b') ... min(s) 'G' 10 ' '
... s[6:10] ... 'a' in s ... max(s) 'Good' False 'y'
We next look at methods in Python that are specific to strings.
Because strings are immutable, sequence modifying operations do not modify the string applied to. Rather, they construct *new* strings that are a modified version of the original.
8.3.3 String Methods
There are a number of methods specific to strings in addition to the general sequence operations. We discuss these methods next.
Checking the Contents of a String

There are times when the individual characters in a string (or substring) needs to be checked. For example, to check whether a character is an appropriate denotation of a musical note, we could do the following,

if char not in ('A', 'B', 'C', 'D', 'E', 'F', 'G'): print('Invalid musical note found')

Checking the Contents of a String			
str.isalpha()	Returns True if *str* contains only letters.	s = 'Hello'	s.isalpha() → True
		s = 'Hello!'	s.isalpha() → False
str.isdigit()	Returns True if *str* contains only digits.	s = '124'	s.isdigit() → True
		s = '124A'	s.isdigit() → False
str.islower() *str*.isupper()	Returns True if *str* contains only lower (upper) case letters.	s = 'hello'	s.islower() → True
		s = 'Hello'	s.isupper() → False
str.lower() *str*.upper()	Return lower (upper) case version of *str*.	s = 'Hello!'	s.lower() → 'hello!'
		s = 'hello!'	s.upper() → 'HELLO!'
Searching the Contents of a String			
str.find(w)	Returns the index of the first occurrence of w in *str*. Returns -1 if not found.	s = 'Hello!'	s.find('l') → 2
		s = 'Goodbye'	s.find('l') → -1
Replacing the Contents of a String			
str.replace(w,t)	Returns a copy of *str* with all occurrences of w replaced with t.	s = 'Hello!'	s.replace('H', 'J') → 'Jello'
		s = 'Hello'	s.replace('ll', 'r') → 'Hero'
Removing the Contents of a String			
str.strip(w)	Returns a copy of *str* with all leading and trailing characters that appear in w removed.	s = ' Hello! ' s = 'Hello\n'	s.strip(' !') → 'Hello' s.strip('\n') → 'Hello'
Splitting a String			
str.split(w)	Returns a list containing all strings in *str* delimited by w.	s = 'Lu, Chao'	s.split(',') → ['Lu', 'Chao']

FIGURE 8-5 String Methods in Python

Since the in operator can also be applied to strings, we can also do the following,
if char not in 'ABCDEFG':
print('Invalid musical note character found') We could take a similar approach for determining if a given character is a lowercase or uppercase letter or digit character; for example,

char in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' or \ letter? char in 'abcdefghijklmnopqrstuvwxyz' or \
char in '0123456789'

Since checking for uppercase/lowercase and digit characters is common in programming, Python provides string methods isalpha, isdigit, isupper, and islower (among others). For example, to perform error checking on an entered credit card number could be done as follows,

if not credit_card.isdigit(): print('Invalid card number')

The method isdigit returns True if and only if each character in string credit_card is a digit. If only *part* of a string is to be checked, then a method can be applied to a slice of a string. For example, if the part numbers of a given company all begin with three letters, a check for invalid part numbers could be done as follows,

if not part_num[0:3].isalpha():
print('Invalid part number')

in which isalpha returns True if and only if the first three characters in part_num are letters. Additional string methods are listed in Figure 8-5. We look at the issue of string search and string modification next.

Python provides a number of methods specific to strings, in addition to the general sequence operations.
Searching and Modifying Strings
String processing involves search. For example, to determine the user name and domain parts of an email address, the ampersand character separating the two would be searched for,

... email_addr 5 'jsmith@somecollege.edu'
... amper_index 5 email_addr.find('@')
... username 5 email_addr[0:amper_index]

... domain 5 email_addr[amper_index 1 1:len(email_addr)] ... print('Username:', username, 'Domain:', domain) Username: jsmith Domain: somecollege.edu

The find method returns the index location of the *fi r s t occurrence* of a specified substring. Since in Python strings are immutable, to update the email address, a *new* string would be constructed with the desired replacement as shown,

... email_addr 5 username 1 '@' 1 'newcollege.edu' ... email_addr jsmith@newcollege.edu

The replace method produces a new string with *every occurrence* of a given substring within the original string replaced with another,

... word 5 'common' ... word 5 'common'
... word.replace('m', 't') ... word 5 word.replace('m', 't') 'cotton' 'cotton'
... word ... word
'common' 'cotton'

Note that for all string modifications, the variable references the same string until it is reassigned. Python also provides a strip method that “strips off” leading and trailing characters from a
string. This is especially useful for stripping off the newline character, \n, from the end of a line in
text processing if needed,

... line 5 'Hello\n' ... line 5 'Hello\n' ... print(line) ... print(line.strip('\n')) 'Hello' 'Hello'

...
...
The find, replace, and strip methods in Python can be used to search and produce modified strings.

8.3.4 Let’s Apply It—Sparse Text Program

Sparse data is data that lacks “density.” A diary containing entries only for holidays and special occasions would contain sparse data. Sparse data can often be compressed. Compressed data should retain all (or most) of the original information. To explore this, the program in Figure 8-7 removes all occurrences of the letter ‘e’ from a provided text file. How much of the compressed text can

be understood indicates how much of the information is retained. This program utilizes the following programming features:

➤ text files ➤ string methods (replace, strip)
Figure 8-6 shows the file produced by the Sparse Text Program for a passage from *Alice's Adventures in Wonderland.*

The main section of the program begins on **line 32**. The program welcome is displayed on **lines 33–34**. In **lines 37–38**, the file name entered by the user is opened for reading and assigned to file object input_file. In **lines 37–40**, a file is opened for writing with the same file name as the input file but with 'e_' added to the beginning and assigned to the file object output_file.

The creation of the modified file is handled by function createModifiedFile (**lines 3–28**) called on **line 44**. The function returns a tuple containing how many occurrences of letter 'e' were removed, and the total character count of the file. The values of the tuple are therefore assigned to variables num_total_char and num_removals. (Note that Python allows such a multiple assignment on lists, tuples, and even on strings.) The input and output files are closed on **lines 47–48**. Finally, both the number of characters removed and the percentage of the file that the removed characters comprised are displayed (**lines 52–54**).

What remains is function createModifiedFile. On **lines 9–11**, variable empty_str is initialized to the empty string, and variables num_total_chars and num_removals are initialized to 0. On **line 16**, variable orig_line_length is set to the length of the currently read line (in variable line) minus one. This is so that the newline character (\n) at the end of the line is not counted.

A Mad Tea-Party (original version)

There was a table set out under a tree in front of the house, and the March Hare and the Hatter were having tea at it: a Dormouse was sitting between them, fast asleep, and the other two were using it as a cushion, resting their elbows on it, and talking over its head. 'Very uncomfortable for the Dormouse,' thought Alice; 'only, as it's asleep, I suppose it doesn't mind.' The table was a large one, but the three were all crowded together at one corner of it: 'No room! No room!' they cried out when they saw Alice coming. 'There's PLENTY of room!' said Alice indignantly, and she sat down in a large arm-chair at one end of the table.

Program Execution ...

```
This program will display the contents of a provided text file with all
occurrences of the letter 'e' removed.

Enter file name (including file extension): alice_tea_party.txt

Thr was a tabl st out undr a tr in front of th hous, and th
March Har and th Hattr wr having ta at it: a Dormous was sitting
btwn thm, fast aslp, and th othr two wr using it as a cushion,
rsting thir lbows on it, and talking ovr its had. 'Vry uncomfortabl
for th Dormous,' thought Alic; 'only, as it's aslp, I suppos it dosn't
mind.' Th tabl was a larg on, but th thr wr all crowdd togthr
at on cornr of it: 'No room! No room!' thy crid out whn thy saw Alic
coming. 'Thr's PLNTY of room!' said Alic indignantly, and sh sat down in
a larg arm-chair at on nd of th tabl

70 occurrences of the letter 'e' removed
Percentage of data lost: 11 %
Modified text in file e_alice_tea_party.txt
```

FIGURE 8-6 Execution of the Sparse Text Program

On **line 20**, variable modified_line is then set to the new string produced by the replace method applied to the current line. Each occurrence of 'e' and 'E' is replaced with the empty string, thus removing these occurrences from the string. Note the consecutive calls to the replace method,

modified_line 5 line.replace('e',empty_str).replace('E',empty_str)

This is possible to do when the first method call returns a value (object) for which the second method call can be applied. In this case, line.replace('e',empty_str) returns a string (consisting of a copy of the original string with all occurrences of the letter 'e' removed), which is then operated on by the second instance of the method call to remove all instances of 'E'.

On **line 21**, the number of characters removed (num_removals) is updated. The number removed from the current line is determined by taking the difference between the original line length (without the newline character) and the length of

the new modified line minus one, so that the newline character in the modified line is not counted. Finally, the modified line of output to the screen (as well as to the output file) is displayed so the user may observe the results of the file processing as it is occurring. A tuple including the number of total characters in the file and the number of removed characters is returned as the function value.

```python
# Sparse Text Program

def createModifiedFile(input_file, outputfile):

    """For text file input_file, creates a new version in file outputfile
        in which all instances of the letter 'e' are removed.
    """

    empty_str = ''
    num_total_chars = 0
    num_removals = 0

    for line in input_file:

        # save original line length
        orig_line_length = len(line) - 1
        num_total_chars = num_total_chars + orig_line_length

        # remove all occurrances of letter 'e'
        modified_line = line.replace('e',empty_str).replace('E',empty_str)
        num_removals = num_removals + \
                        (orig_line_length - (len(modified_line)-1))

        # simulataneouly output line to screen and output file
        print(modified_line.strip('\n'))
        output_file.write(modified_line)

    return (num_total_chars, num_removals)

# --- main

# program welcome
print("This program will display the contents of a provided text file")
print("with all occurrences of the letter 'e' removed.\n”)

# open files for reading and writing
file_name = input('Enter file name (including file extension): ')
input_file = open(file_name,'r')
new_file_name = 'e_' + file_name
output_file = open(new_file_name,'w')

# create file with all letter e removed
print()
num_total_chars, num_removals = createModifiedFile(input_file, output_file)

# close current input and output files
input_file.close()
output_file.close()

# display percentage of characters removed
print()
print(num_removals, "occurrences of the letter 'e' removed")
print('Percentage of data lost:',
      int((num_removals / num_total_chars) * 100), '%')
print('Modified text in file', new_file_name)
```

FIGURE 8-7 Sparse Text Program

Self-Test Questions

1. Some string methods alter the string they are called on, while others return a

new altered version of the string. (TRUE/FALSE)
2. The find method returns the number of occurrences of a character or substring within a given string. (TRUE/FALSE)
3. Which of the results below does s[2:4] return for the string s = 'abcdef'. **(a)** 'cd' **(b)** 'bcd' **(c)** 'bc' **(d)** 'cde'

4. Indicate which of the following is true.
(a) String method isdigit returns true if the string applied to contains any digits.
(b) String method isdigit returns true if the string applied to contains only digits.

5. Indicate which of the following s.replace('c','e')returns for s 5'abcabc'. **(a)** 'abeabc' **(b)** 'abeabe'

6. Which of the results below does s.strip('-')return for the string s 5'---ERROR---'.
(a) '---ERROR' **(b)** 'ERROR---' **(c)** 'ERROR'

ANSWERS: 1. False, 2. False, 3. (a), 4. (b), 5. (b), 6. (c)

8.4 Exception Handling

Various error messages can occur when executing Python programs. Such errors are called *exceptions*. So far we have let Python handle these errors by reporting them on the screen. Exceptions can be “caught” and “handled” by a program, however, to either correct the error and continue execution, or terminate the program gracefully. We take this opportunity to discuss the fundamentals of exception handling in the use of text files.

8.4.1 What Is an Exception?

An **exception** is a value (object) that is *raised* (“thrown”) signaling that an unexpected, or “exceptional,” situation has occurred. Python contains a predefined set of exceptions referred to as **standard exceptions**. We list some of the standard exceptions in Figure 8-8.

ImportError	raised when an import (or from…import) statement fails
IndexError	raised when a sequence index is out of range
NameError	raised when a local or global name is not found
TypeError	raised when an operation or function is applied to an object of inappropriate type
ValueError	raised when a built-in operation or function is applied to an appropriate type, but of inappropriate value
IOError	raised when an input/output operation fails (e.g., “file not found”)

FIGURE 8-8 Some Standard Exceptions in Python

The standard exceptions are defined within the exceptions module of the Python Standard Library, which is automatically imported into Python programs. We have seen a number of these exceptions before in our programming,

... lst 5 [1, 2, 3]
... lst[3]
Traceback (most recent call last):

File " ,pyshell#4.", line 1, in ,module. lst[3]
IndexError: list index out of range ... 2 1 '3'
Traceback (most recent call last):

File " ,pyshell#7.", line 1, in ,module. 2 1 '3'
TypeError: unsupported operand type(s) for 1: 'int' and 'str' ... lsst[0]
Traceback (most recent call last):
File ",pyshell#5.", line 1, in ,module. lsst[0]
NameError: name 'lsst' is not defined

... int('12.04')

Traceback (most recent call last): File ",pyshell#9.", line 1, in ,module.
int('12.04')

ValueError: invalid literal for int() with base 10: '12.04'

Raising an exception is a way for a function to inform its client a problem has occurred that *the function itself cannot handle*. For example, suppose a function called getYN prompts a user to enter 'y' or 'n'. If the user enters something other than these two values, the function can simply prompt the user to re-enter. On the other hand, if a function called isEven is to be passed a numeric value, but is passed a string instead, it cannot correct the problem. It can only notify the client of the problem, leaving it up to the client to determine what to do. We next discuss in detail the raising and catching of exceptions.

An **exception** is a value (object) that is “raised” by a function signaling that an unexpected, or “exceptional,” situation has occurred that the function itself cannot handle.

8.4.2 The Propagation of Raised Exceptions

Raised exceptions are not required to be handled in Python. When an exception is raised and not handled by the client code, it is automatically propagated back to the *client’s* calling code (and *its* calling code, etc.) until handled. If an exception is thrown all the way back to the top level (main module) and not handled, then the program terminates and displays the details of the exception as depicted in Figure 8-9.

For example, a program might not be able to determine whether a password is valid at the source of the input. Rather, the password must be verified in a password file that is accessed only after a *chain* of function calls have been made. If the password is found invalid, an exception is propagated from the function identifying the error all the way back to the function responsible for user input, which can then prompt the user to re-enter their password.

FIGURE 8-9 The Propagation of Exceptions

An exception is either handled by the client code, or automatically propagated back to the client’s calling code, and so on, until handled. If an exception is thrown all the way back to the main module (and not handled), the program

terminates displaying the details of the exception.

8.4.3 Catching and Handling Exceptions

Many of the functions in the Python Standard Library raise exceptions. For example, the factorial function of the Python math module raises a ValueError exception when a negative value is passed to it, as shown in Figure 8-10.

```
import math

num = int(input('Enter number to compute factorial of: '))
print('The factorial of', num, 'is', math.factorial(num))
```

```
Enter number to computer factorial of: -5
The factorial of -5 is
Traceback (most recent call last):
  File "C:\My Python Programs\exception_example1.py", line 4, in <module>
    print 'The factorial of', num, 'is', math.factorial(num)
ValueError: factorial() not defined for negative values
```

FIGURE 8-10 Simple Program without Exception Handling

When 25 is passed to the factorial function, an exception is raised, thrown back to the client code to catch and handle. Since the client code (here, the code in the main module) does not attempt to catch the exception, the exception is caught by the Python interpreter, causing the program to terminate and display an error message indicating the exception type. The last line of the message indicates that a ValueError exception occurred within the factorial function. The previous lines indicate where in the client code this function was called (line 4). In Figure 8-11 we give a version of the program that catches the raised exception.

```
import math

num = int(input('Enter number to compute factorial of: '))
try:
    print(math.factorial(num))
except ValueError:
    print('Cannot compute the factorial of negative numbers')
```

```
Enter number to compute factorial of: -5
Cannot compute the factorial of negative numbers
```

FIGURE 8-11 Simple Program with Exception Handling

The call to the factorial function is contained within a **try suite** (**try block**)—the block of code surrounded by try and except headers. The suite following the except header is referred to as an **exception handler**. This exception header "catches" exceptions of type ValueError. The exception handler in this case simply outputs an error message to the user before terminating the program. This program termination is much more user-friendly than the previous version.

Finally we show a version of the program that recovers from the exception. Rather than terminating the program, the user is prompted again for input so that the program can continue executing. This new version is given in Figure 8-12.

```
import math

num = int(input('Enter number to compute factorial of: '))
valid_input = False

while not valid_input:
    try:
        result = math.factorial(num)
        print(result)
        valid_input = True
    except ValueError:
        print('Cannot compute factorial of negative numbers')
        num = int(input('Please re-enter: '))
```

```
Enter number to compute factorial of: -5
Cannot compute the factorial of negative numbers
Please re-enter: 5
120
```

FIGURE 8-12 Program Recovery via Exception Handling

Note that within a try suite in Python, any statement making a call (either directly or indirectly) to a function that raises an exception causes the rest of the statements in the suite to be skipped, as depicted in the figure.
Exceptions are caught and handled in Python by use of a **try block** and **exception handler**.

8.4.4 Exception Handling and User Input
Besides exceptions raised by built-in functions, programmer-defined functions may raise exceptions as well. Suppose we prompted the user to enter the current month as a number,
month 5 input('Enter current month (1–12): ') The input function will return

whatever is entered as a string. We can do integer type conversion on this value to make it an integer type,
month 5 int(input('Enter current month (1–12): '))

If the input string contained non-digit characters (except for 1and 2), the int function would raise a ValueError exception. However, there also needs to be a check for values outside the range 1–12. This can be done as shown in Figure 8-13.

```
valid = False

while not valid:
    try:
        month = int(input('Enter current month (1-12): '))

        while month < 1 or month > 12:
            print('Invalid Input - Must be in the range 1-12'
            month = int(input('Enter current month (1-12): '))
        valid = True
    except ValueError:
        print('Invalid Month Value')
```

FIGURE 8-13 Input Error Checking (Version 1)

Athough this works, a better approach is to design a function called getMonth that raises a ValueError exception for either error condition—if the user enters non-digit characters, or if the user enters a numeric value outside the range 1–12. Such a function is given in Figure 8-14.

```
def getMonth():
    month = int(input('Enter current month (1-12): '))

    if month < 1 or month > 12:
        raise ValueError('Invalid Month Value')

    return month
```

FIGURE 8-14 Programmer-Defined Function with Raised Exception
In this version of getMonth, if non-digit characters are entered, the ValueError exception is automatically raised by the built-in int type conversion function,

... getMonth()
Enter current month (1–12): 1a
Traceback (most recent call last):

File " ,pyshell#18 .", line 1, in ,module .
getMonth()
File " C:\My Python Programs\exception /Raising Excpt Test.py", line 2, in getMonth
month 5 int(input('Enter current month (1–12): ')) ValueError: invalid literal for int() with base 10: '1a'

Since there is no try block around the input assignment statement, the exception is not caught and therefore is thrown back to the Python interpreter. If a valid numeric string is entered, but the value is outside the range 1–12, then a ValueError is raised by function getMonth and is thrown back to the client,

... getMonth()
Enter current month (1–12): 14
Traceback (most recent call last):

File " ,pyshell#23 .", line 1, in ,module .
getMonth()
File " C:\My Python Programs\exception /Raising Excpt Test.py", line 5, in getMonth
raise ValueError('Invalid Month Value')
ValueError: Invalid Month Value

Note that raised exceptions can optionally have a description that further describes the error, as has been done here,
raise ValueError('Invalid Month Value')

This error message is displayed when the ValueError generated is the one thrown by function getMonth. With this function, the less elegant error checking code in Figure 8-13 can now be replaced with that in Figure 8-15.

```
valid = False

while not valid:
    try:
        month = getMonth()
        valid = True
    except ValueError:
        print('Invalid Month Entry\n')
```

FIGURE 8-15 Input Error Checking (Version 2)
In this case, when an invalid input is entered, we get the following,

Enter current month (1–12): 1a Invalid Month Value
Enter current month (1–12): 14 Invalid Month Value

This is a much cleaner error reporting for the user. The error message Invalid Month Value, however, doesn't indicate *why* the input was invalid (that is, either that an invalid character was found, or an integer outside the range 1–12 was entered). We could go one step further and output the error message associated with the specific ValueError exception thrown, as shown in Figure 8-16.

```
valid = False

while not valid:
    try:
        month = getMonth()
        valid = True
    except ValueError as err_mesg:
        print(err_mesg,'\n')
```

FIGURE 8-16 Input Error Checking (Version 3)

The line except ValueError as err_mesg not only catches the ValueError exceptions, but with as *identifier* added (here as err_mesg), the error message of the particular ValueError exception is assigned to the specified identifier name,

Enter current month (1–12): 1a
invalid literal for int() with base 10: '1a' Enter current month (1–12): 14
Invalid Month Value
Enter current month (1–12): 12
...
We next look at exception handling related to file processing.
Programmer-defined functions may raise exceptions in addition to the exceptions raised by the built-in functions of Python.
8.4.5 Exception Handling and File Processing

We saw that when opening a file for reading, an exception is raised if the file cannot be found. In this case, the standard IOError exception is raised and the program terminates with a 'No such file or directory' error message. We can catch this exception and handle the error as shown in Figure 8-17.

```
file_name = input('Enter file name: ')
empty_str = ''

input_file_opened = False

while not input_file_opened:
    try:
        input_file = open(file_name, 'r')
        input_file_opened = True

        line = input_file.readline()

        while line != empty_str:
            print(line.strip('\n'))
            line = input_file.readline()
    except IOError:
        print('File Open Error\n')
        file_name = input('Enter file name: ')
```

FIGURE 8-17 Exception Handling of Open File Error

V ariable file_name stores the file name entered by the user. Variable input_file_opened is initialized to False. The while loop continues to iterate as long as input_file_opened is False. Because of the try block within the loop, every time that open(file_name, 'r') raises an exception, the remaining lines in the try block are skipped, and the exception handler following the except header is executed. Only when the call to open does not throw an exception do all the instructions in the try block get executed, with the program continuing after the while loop. A similar, but much less likely exception can be raised when opening a file for writing and the file system (hard disk, for example) we want to write to is full.

Note that when reading from a text file, the readline method does not raise any exceptions. When the end of file is reached, readline returns an empty string rather than throwing an exception.
IOError exceptions raised as a result of a file open error can be caught and handled.

8.4.6 Let’s Apply It—Word Frequency Count Program

The following Python program (Figure 8-19) prompts the user for the name of a text file to open and a word to search for, and displays the number of times that the word occurs within the file. This program utilizes the following Python programming features:

➤ text files/readline()
➤ string methods lower(), index()
Example execution of the program is given in Figure 8-18.

Program execution begins at **line 77**. First, the program welcome is displayed. On **line 81**, function getFile is called, which prompts the user for the file name to open for reading. It returns a tuple containing both the file name and the associated input file object, assigned to variables file_name and input_file, respectively.

Text File: wordtest.txt

```
This is a sample text file.
The word 'the' in this file is to be counted.
Must be careful not to count words like their and there.
Should only count the word "the" when it is a separate word.
This is the last line of the file.
```

Program Execution ...

```
This program will display the number of occurrences of a
specified word within a given text file

Enter input file name: word
Input file not found - please reenter

Enter input file name: wordtest.txt
Enter word to search: apple
No occurrences of word 'apple' found in file wordtest.txt
```

Program Execution ...

```
This program will display the number of occurrences of a
specified word within a given text file

Enter input file name: wrdtest.txt
Input file not found - please reenter

Enter input file name: wordtest.txt
Enter word to search: the
The word 'the' occurs 6 times in file wordtest.txt
```

FIGURE 8-18 Execution of the Word Frequency Count Program

On **line 84** the user is prompted for the word to search for, stored in variable search_word. In the following line, search_word is reassigned to all lowercase characters by use of method lower(). The file lines read in function countWords are also converted to lowercase so that the matching of words does not depend

on whether letters are in uppercase or lowercase (in other words, so that it is not *case sensitive*).

The function that does most of the work, function countWords, is called on **line 88**. It is passed the file object, input_file, as well as the words to search for. The function returns, as an integer, the number of occurrences found. Finally, in **lines 91–98**, the results of the search are displayed.

The getFile function called from the main section of the program prompts the user for a file name to open (**line 12**). The file is opened for reading on **line 13**. If the file is not found, an IOException is raised and caught by the except clause on **line 15**. In this case, an error message is printed (**line 16**) and the while loop at **line 10** iterates again, prompting the user to re-enter. If an exception is not thrown, then **line 14** is executed setting input_file_opened to True, causing the loop to terminate and the function to return both the file name and input file object as a tuple.

Function countWords, the final function of the program, is on **lines 21–72**. It is passed the input_file object and the search word, and returns the number of occurrences of the search word within the file. **Lines 28–31** perform the initialization for the function. Variable space is assigned to a string containing only a single blank character (to aid in the readability of the program). Variable num_occurrences, which keeps count of the number of times the search word appears in the file, is initialized to 0. And word_delimiters is set to a tuple

```
# Word Frequency Count Program

def getFile():

    """Returns the file name and associated file object for reading
       the file as a tuple of the form (file_name, input_file).
    """

    input_file_opened = False
    while not input_file_opened:
        try:
            file_name = input('Enter input file name: ')
            input_file = open(file_name, 'r')
            input_file_opened = True
        except IOError:
            print('Input file not found - please reenter\n')

    return (file_name, input_file)


def countWords(input_file, search_word):

    """Returns the number of occurrences of search_word in the
       provided input_file object.
    """

    # init
    space = ' '
    num_occurrences = 0
    word_delimiters = (space, ',', ';', ':', '.','\n',
                       '"',"'", '(', ')')

    search_word_len = len(search_word)

    for line in input_file:
        end_of_line = False

        # convert line read to all lower case chars
        line = line.lower()

        # scan line until end of line reached
        while not end_of_line:
            try:
                # search for word in current line
                index = line.index(search_word)

                # if word at start of line followed by a delimiter
                if index == 0 and line[search_word_len] in
                   word_delimiters:
                        found_search_word = True

```

FIGURE 8-19 Word Frequency Count Program (*Continued*)

containing all the possible word delimiters—a space character, a comma, a semicolon, a colon, a period, and a newline character.

Because the length of the search word is used multiple times, it is calculated

once and stored in variable search_word_len (**line 33**). The for loop on **line 35** reads each line of the file. Each line is scanned for all occurrences of the search word. Variable end_of_line is initialized to

```
53                  # if search word within line, check chars before/after
54                  elif line[index - 1] in word_delimiters and \
55                     line[index + search_word_len] in word_delimiters:
56                      found_search_word = True
57
58                  # if found within other letters, then not search word
59                  else:
60                      found_search_word = False
61
62                  # if search word found, increment count
63                  if found_search_word:
64                       num_occurrences = num_occurrences + 1
65
66                  # reset line to rest of line following search word
67                  line = line[index + search_word_len: len(line)]
68
69              except ValueError:
70                  end_of_line = True
71
72      return num_occurrences
73
74  # ---- main
75
76  # program welcome
77  print('This program will display the number of occurrences of a')
78  print('specified word within a given text file\n')
79
80  # open file to search
81  file_name, input_file = getFile()
82
83  # get search word
84  search_word = input('Enter word to search: ')
85  search_word = search_word.lower()
86
87  # count all occurrences of search word
88  num_occurrences = countWords(input_file, search_word)
89
90  # display results
91  if num_occurrences == 0:
92      print('No occurrences of word', "'" + search_word + "'",
93            'found in file', file_name)
94  else:
97      print('The word', "'" + search_word + "'", 'occurs',
98            num_occurrences, 'times in file', file_name)
```

FIGURE 8-19 Word Frequency Count Program

False (**line 36**) for the while loop below it. On **line 39**, all characters in the line are converted to lowercase to allow for the matching of words of different case (mentioned above). Then, beginning on **line 42**, a while loop is used to search for all occurrences of the search word in the current line, continuing until the end of line is found.

On **line 45**the index string method is called on the current line to search for the first occurrence of the search word. It therefore performs the same task as the find method. However, whereas find returns a 21 if the string is not found, the index method raises a ValueError exception instead. Since we have now introduced exception handling in Python, we here use the index method.

The if statement on **line 48** checks if the search string matches the beginning characters of the line (followed by a delimiter). This determines if the first word of the current line matches the search string. If so, then found_search_string is set to True (**line 50**). Otherwise, a check is made on **line 54** to see if the matched string in the current line is immediately preceded and followed by a delimiter. If found, then found_search_string is also set to True, otherwise, found_ search_string is set to False. On **lines 63–64**, if found_search_word is True, num_ occurences is incremented by 1.

V ariable line is reassigned to the substring following the last word found (**line 67**). This is needed because the index method only finds the *fi r s t* occurrence of a given substring within a string, and not *all* occurrences. Therefore, the line is continually shortened and scanned until no further instances of the word are found within the line. Once the complete line is scanned, the while loop terminates and control is returned to the top of the for loop (on **line 35**). When there are no more lines of the file to read, execution continues at **line 72** and the value of num_ occurrences is returned.

Self-Test Questions
1. An exception is,
(a) an object **(b)** a standard module **(c)** a special function
2. Which of the following is not a standard exception in Python?
(a) ValueError **(b)** AssignmentError **(c)** NameError **(d)** IOError **3.** The standard exceptions are automatically imported into Python programs. (TRUE/FALSE)
4. All raised standard exceptions must be handled in Python. (TRUE/FALSE) **5.** Which one of the following is true?

(a) When calling the built-in open function for file handling, it *must* be called from within a try block with an appropriate exception handler.
(b) When calling the built-in open function for file handling, it *should* be called from within a try block with an appropriate exception handler.

6. In addition to catching standard exceptions in Python, a program may also

raise standard exceptions. (TRUE/FALSE)

ANSWERS: 1. (a), 2. (b), 3. True, 4. False, 5. (b), 6. True

COMPUTATIONAL PROBLEM SOLVING

8.5 Cigarette Use/Lung Cancer Correlation Program

In this section, we design a program using data from the Centers for Disease Control and Prevention (CDC) for computing the correlation between cigarette use and incidences of lung cancer (Figure 8-20). One data set gives the percentage of the population that smoke cigarettes within the United States by state. The other gives the rate of lung cancer per 100,000 individuals by state. The computed correlation, along with other factors, can be used to determined if there is a causal relationship between the two.

8.5.1 The Problem

The problem is to calculate a correlation value for two sets of data. Correlation is measured on a scale of 21 to 1, with 1 indicating a *perfect positive correlation*, and 21 a *perfect negative correlation*. For example, there is a positive correlation between the amount of ice cream sold and the current temperature—as temperatures rise, so do the sales of ice cream. There is a negative correlation between the number of snow blowers sold and the current temperature—as temperatures rise, the number of snow blowers sold decreases.

In a perfect correlation, knowing one value allows one to determine the *exact* value of the other. In real-world situations, perfect correlations are almost never found. For example, many other factors can affect exactly how much ice cream is sold—there may be a truckers' strike reducing deliveries, there may be a recall of a certain brand of ice cream, and so forth. Therefore, most correlation values fall somewhere between 21 and 1.

FIGURE 8-20
Adverse Effects of Cigarette Smoking

8.5.2 Problem Analysis

In the two sets of data provided in Figure 8-21, one contains the percentage of the population that smokes state-by-state in the United States, and the other the state-by-state rates of incidence of lung cancer per 100,000 individuals.

The mathematical formula for computing correlation is given below.

$$r = \frac{N\sum xy - \sum x \sum y}{\sqrt{(N\sum x^2 - (\sum x)^2)(N\sum y^2 - (\sum y)^2)}}$$

where,
N is equal to the number of pairs of values in the data
x and y are a given pair of values
$\sum xy$ is the sum of the products of paired scores
$\sum x$ is the sum of the scores of one data set
$\sum y$ is the sum of the scores of the other data set

$\sum x^2$ is the sum of the squares of the scores of one data set
$\sum y^2$ is the sum of the squares of the scores of the other data set

For our data, N is equal to 48 (all states except Arizona and Wisconsin, which were not provided in the data sets from the CDC). An example of x and y values are the two values for Alabama, x 5 23.3 (percent of population that smokes) and y 5 75.1 (lung cancer cases per 100,000 individuals). Note that to apply this formula, the values of x and y do not have to be in the same units.

```
State, Percent Cigarette Smokers
Alabama,23.3
Alaska,24.2
Arkansas,23.7
California,14.9
Colorado,17.9
Connecticut,17
Delaware,21.7
Florida,21
Georgia,20
Hawaii,17.5
Idaho,16.8
Illinois,20.5
Indiana,24.1
Iowa,21.5
Kansas,20
Kentucky,28.6
Louisiana,23.4
Maine,20.9
Maryland,17.8
Massachusetts,17.8
Michigan,22.4
Minnesota,18.3
Mississippi,25.1
Missouri,23.3
Montana,19
Nebraska,18.6
Nevada,22.2
New Hampshire,18.7
New Jersey,18.1
New Mexico,20.2
New York,18.3
North Carolina,22.1
North Dakota,19.6
Ohio,22.5
Oklahoma,25.1
Oregon,18.5
Pennsylvania,21.5
Rhode Island,19.3
South Carolina,22.3
South Dakota,20.4
Tennessee,22.6
Texas,18.1
Utah,9.8
Vermont,18
Virginia,19.3
Washington,17.1
West Virginia,25.7
Wyoming,21.6
```

CDC Data on the Percentage of the Population that Smoke Cigarettes within the U.S. by State (2006)

```
Cases Lung Cancer per 100,000
Alabama,75.1
Alaska,69.8
Arkansas,77.7
California,51.6
Colorado,48.4
Connecticut,68.3
Delaware,83.5
Florida,68.7
Georgia,71
Hawaii,51.7
Idaho,54
Illinois,71.4
Indiana,77.2
Iowa,67.1
Kansas,68.4
Kentucky,97.2
Louisiana,77.5
Maine,80.1
Maryland,62.9
Massachusetts,64.5
Michigan,72
Minnesota,55.8
Mississippi,77.1
Missouri,77.9
Montana,60.8
Nebraska,61.3
Nevada,73.3
New Hampshire,66.8
New Jersey,64.5
New Mexico,43.9
New York,63
North Carolina,75.3
North Dakota,53
Ohio,72.8
Oklahoma,80.8
Oregon,64.4
Pennsylvania,70
Rhode Island,69.5
South Carolina,70.7
South Dakota,59.2
Tennessee,82.2
Texas,62
Utah,28
Vermont,79.2
Virginia,65.8
Washington,65.2
West Virginia,91
Wyoming,48.9
```

CDC Data on the Rate of Lung/Bronchus Cancer per 100,000 Individuals within the U.S. by State (2006)

FIGURE 8-21 Cigarette Smoking and Incidence of Lung Cancer Data

8.5.3 Program Design

The data is provided in two comma-separated (CSV) files. Thus, we must first

read the data from these file. Using this data, we must then compute the correlation of the two sets of data on a scale of 21 (a perfect negative correlation) to 1 (a perfect positive correlation).

Meeting the Program Requirements

The program must only display the correlation value between the two sets of data. Since the program requirements do not specify how the result is to be presented (for example, on a scale), the raw numerical value will simply be displayed on the screen.

Data Description

The data from the files will be stored in two lists so that the values can be easily accessed during the calculation of the correlation. Thus, the data is simply organized as two corresponding “parallel” lists of floating-point values.

Algorithmic Approach

The algorithm for this program is the means of calculating the correlation values using the mathematical formula given above.

Overall Program Steps

The overall steps in the program design are given in Figure 8-22.

FIGURE 8-22 Overall Steps of the Smoking/
Lung Cancer Correlation Program

Modular Design

The modular design for this program is given in Figure 8-23.

FIGURE 8-23 Modular Design of the Smoking/Lung Cancer Correlation Program

Following this modular design, there are three functions in the program: openFiles, readFiles, and calculateCorrelation. The main module utilizing these functions is given in Figure 8-24.

```
# ---- main

# program greeting
print('This program will determine the correlation between')
print('cigarette smoking and incidences of lung cancer\n')

try:
    # open data files
    smoking_datafile, cancer_datafile = openFiles()

    # read data
    smoking_data, cancer_data = readFiles(smoking_datafile, cancer_datafile)

    # calculate correlation value
    correlation = calculateCorrelation(smoking_data, cancer_data)

    # display correlation value
    print('r_value = ', correlation)
except IOError as e:
    print(str(e))
    print('Program terminated ...')
```

FIGURE 8-24 Main Module of the Smoking/Lung Cancer Correlation Program Design

8.5.4 Program Implementation and Testing

We develop and test the program in stages by developing and unit testing each of the three functions of the modular design. We begin with function openFiles.

Development and Unit Testing of Function **openFiles**

In the main module, we see that openFiles is designed as a value-returning function with no arguments, returning file objects for each of the opened data

files. An implementation of this function is given in Figure 8-25.

```
1  # Module module_openFiles
2
3  def openFiles():
4      """
5          Prompts the user for the file names to open, opens the files, and
6          returns the file objects for each in a tuple of the form
7          (smoking_datafile, cancer_datafile).
8
9          Raises an IOError exception if the files are not successfully
10         opened after four attempts of entering file names.
11     """
12     # init
13     smoking_datafile_opened = False
14     cancer_datafile_opened = False
15     num_attempts = 4
16
17     # prompt for file names and attempt to open files
18     while ((not smoking_datafile_opened) or (not cancer_datafile_opened)) \
19            and (num_attempts > 0):
20         try:
21             if not smoking_datafile_opened:
22                 file_name = input('Enter smoking data file name: ')
23                 smoking_datafile = open(file_name, 'r')
24                 smoking_datafile_opened = True
25
26             if not cancer_datafile_opened:
27                 file_name = input('Enter lung cancer data file name: ')
28                 cancer_datafile = open(file_name, 'r')
29                 cancer_datafile_opened = True
30         except IOError:
31             print('File not found:', file_name + '.', 'Please reenter\n')
32             num_attempts = num_attempts - 1
33
34     # if one of more file not opened, raise IOError exception
35     if not smoking_datafile_opened or not cancer_datafile_opened:
36         raise IOError
37     else:
38         return (smoking_datafile, cancer_datafile)
```

FIGURE 8-25 Implementation of Function openFiles

Function openFiles is initially put in its own Python file so that it can be easily imported for unit testing. The function handles the task of inputting from the user the data file names (for both the smoking-related data file and the lung cancer–related data file), opening the files for reading, and returning the two file objects created as a tuple of the form (smoking_datafile, cancer_ datafile).

In the init section, variables smoking_datafile_opened and cancer_datafile_ opened are initialized to False (**lines 13–14**). They are used in the control of the

following while loop so that the user is continually prompted until correct file names for the data files are entered. On **line 15**, variable num_attempts is initialized to 4. This variable is decremented each time that an invalid file name (for either file) is entered. If the counter reaches 0, the while loop terminates and the final if statement checks if the files have been successfully opened. If not, then an IOError exception is raised, terminating the function.

The while loop at **line 18** continues to iterate as long as either of the two data files has not been successfully opened, and as long as the value of variable num_attempts is greater than zero. The statements in the loop are contained within a try block (**lines 21–29**), used to catch any IOError exceptions raised when attempting to open either of the two data files by call to method open,

smoking_datafile 5 open(file_name, 'r')
cancer_datafile 5 open(file_name, 'r')

Following the calls to method *open* for the smoking data file (**line 23**) and the cancer data file (**line 28**) is an assignment statement that assigns smoking_datafile_opened and cancer_datafile_ opened, respectively, to True. Since these statements would not be reached if the call to the open method raised an exception, these Boolean variables provide a means of determining which, if any, of the two files has been opened successfully. If either call to method open raises an exception, it is caught by the except clause (**line 30**). As a result, on **line 31** the error message 'File not found: *filename*. Please reenter', for the *filename* in variable file_name is displayed. In addition, variable num_attempts is decremented by 1 (**line 32**), and control returns to the top of the while loop. Finally, when the loop terminates with both files successfully opened, a tuple containing each of the file objects is returned (**line 38**).

Function openFiles is not easily tested interactively in the Python shell. Therefore, we develop a *test driver* program that simply calls function openFile (imported from module_ openFiles) to open the data files, and displays the first line of each file on the screen. The test driver program is given in Figure 8-26.

```
# TEST DRIVER for Function openFiles of Smoking/Cancer Correlation Program

from module_openFiles import *

empty_str = ''
print('TESTING FUNCTION openFiles')

try:
    # open data files
    smoking_datafile, cancer_datafile = openFiles()

    # display sucessfully opened
    print('Data files successfully opened')

    # display first line of smoking data file
    print('\nReading first line of smoking data file ...')
    line = smoking_datafile.readline()
    print(line.strip('\n'))

    # display first line of cancer data file
    print('\nReading first line of cancer data file ...')
    line = cancer_datafile.readline()
    print(line.strip('\n'))
except IOError:
    print('Too many attempts of opening input files')
    print('Program terminated ...')
```

FIGURE 8-26 Test Driver for Function openFiles

Since function openFiles is designed to raise an IOError exception if the files are not successfully opened after four attempts, we test this aspect of the program by purposely entering incorrect file names four times. The result of this testing is given in Figure 8-27.

```
TESTING FUNCTION openFiles
Enter smoking data file name: CDC_Cigarette_Smking_Data.csv
File not found: CDC_Cigarette_Smking_Data.csv. Please reenter

Enter smoking data file name: CDC_Cigarette_Smokin_Data.csv
File not found: CDC_Cigarette_Smokin_Data.csv. Please reenter

Enter smoking data file name: CDC_Cigarette_Smoking_Data.csv
Enter file name with lung cancer data: CDC_Lng_Cancer_Data.csv
File not found: CDC_Lng_Cancer_Data.csv. Please reenter

Enter lung cancer data file name: CDC_Lung_Caner_Data.csv
File not found: CDC_Lung_Caner_Data.csv. Please reenter

Too many attempts of reading input file
Program terminated ...
>>>
```

FIGURE 8-27 Invalid File Names Test Case for Function openFiles

Thus, this aspect of the function appears to be working. We next give it correct file names to see if the function successfully opens and correctly displays the first line of each file. The results are shown in Figure 8-28.

```
TESTING FUNCTION openFiles
Enter smoking data file name: CDC_Cigarette_Smoking_Data.csv
Enter lung cancer data file name: CDC_Lung_Cancer_Data.csv
Data files successfully opened

Reading first line of smoking data file ...
State, Percent Cigarette Smokers , CDC 2006 Data (except AZ, WI)

Reading first line of cancer data file ...
Cases Lung Cancer per 100000 CDC Data 2006 (except AZ,WI)
>>>
```

FIGURE 8-28 Valid File Names Test Case for Function openFiles

The output is as expected. We can assume that function openFiles is implemented correctly. We therefore unit test function readFiles next.

Development and Unit Testing of Function **readFiles**

As done for function openFiles, we put function readFiles in its own file to be imported by its test driver, named module_readFiles.py. An implementation of function readFiles, implemented as a separate module, is given in Figure 8-29.

```
1  # Module module_readFiles
2
3  def readFiles(smoking_datafile, cancer_datafile):
4
5       """Reads the data from the provided file objects smoking_datafile
6          and cancer_datafile. Returns a list of the data read from each
7          in a tuple of the form ([smoking data], [cancer data]).
8      """
9
10     # init
11     smoking_data = []
12     cancer_data = []
13     empty_str = ''
14
15     # read past file headers
16     smoking_datafile.readline()
17     cancer_datafile.readline()
18
19     # read data files
20     eof = False
21
22     while not eof:
23
24         # read line of data from each file
25         s_line = smoking_datafile.readline()
26         c_line = cancer_datafile.readline()
27
28         # check if at end-of-file of both files
29         if s_line == empty_str and c_line == empty_str:
30             eof = True
31
32         # check if end of smoking data file only
33         elif s_line == empty_str:
34             raise IOError('Unexpected end-of-file: smoking data file')
35
36         # check if at end of cancer data file only
37         elif c_line == empty_str:
38             raise IOError('Unexpected end-of-file: cancer data file')
39
40         # append line of data to each list
41         else:
42             smoking_data.append(s_line.strip().split(','))
43             cancer_data.append(c_line.strip().split(','))
44
45     # return list of data from each file
46     return (smoking_data, cancer_data)
```

FIGURE 8-29 Implementation of Function readFiles

Function readFiles parameters are passed the file objects of the smoking-related and cancer-related data files (**line 3**). It reads the data from each file and returns it in two lists. On **lines 11–12**, variables smoking_data and cancer_data are

initialized to an empty list, and on **line 13**variable empty_str is assigned to an empty string. The first line is then read from each file (**line 16–17**), which contain file headers (descriptions of the file contents),

State, Percent Cigarette Smokers Cases Lung Cancer per 100,000
(header of smoking-related data file) (header of cancer-related data file)

On **line 20**, variable eof is set to False. The value of this variable indicates whether the end- of-file has been reached for the files. The while loop at **line 22** continues to iterate as long as eof is False. A line from each file is then read (**line 25–26**) and stored in variables s_line (from the smoking-related data file) and c_line (from the cancer-related data file). The following if statement (**line 29**) checks if each of these variables is equal to the empty string. If true, then the end of file of each has been reached, and thus variable eof is assigned to True (**line 30**) If not, then a check is made (on **line 33** and **line 37**) if either s_line or c_line is equal to the empty string. If so, then the end of one file has been reached before the other. In that case, an IOError exception is raised (on **line 34** or **38**) with one of the following error message strings,

'Unexpected end-of-file: smoking data file' or
'Unexpected end-of-file: cancer data file'

If the strings in variables s_line and c_line are each found non-empty, then on **lines 42–43** they are appended to lists smoking_data and cancer_data, respectively. Since each line returned by readLine contains a final newline character, method strip is used to strip off the last character of each line. Also, because each line of the files is of the form

state_name, data_value
the split string method is used to split each string into two values (state_name and data_ value) using the comma as the separation character. This returns the two values in a list of the form [*state_name*, *data_value*]
Finally, when all the lines of the files have been read, the data in lists smoking_data and cancer_data are returned as a tuple.
As we did for function openFile, we develop a simple test driver program for testing function readFiles. The test driver (in Figure 8-30) first prompts the user for each of the file names

```
# TEST DRIVER for Function readFiles of Smoking/Cancer Correlation Program

from module_readFiles import *

try:
    file_name = input('Enter file name with smoking data: ')
    smoking_datafile = open(file_name, 'r')

    file_name = input('Enter file name with lung cancer data: ')
    cancer_datafile = open(file_name, 'r')

    smoking_data, cancer_data = readFiles(smoking_datafile, cancer_datafile)

    print('\nList of smoking data read')
    print(smoking_data)
    print('\nList of cancer data read')
    print(cancer_data)
except IOError as e:
    print(str(e))
```

FIGURE 8-30 Test Driver for Function readFiles

without doing any I/O error checking. This is because in the complete program version, function openFile will handle this, and therefore we can alleviate the test driver of that. Thus, the files are opened (assuming that correct file name are given) and the contents read and displayed on the screen.

Function readFiles raises an exception if the two data files do not have the same number of lines. Therefore, we test for that condition by purposely entering the names of two data files in which one file is shorter than the other. The results of this test case is given in Figure 8-31.

```
Enter file name with smoking data: CDC_Cigarette_Smoking_Data.csv
Enter file name with lung cancer data: CDC_Lung_Cancer_Data_Short.csv
Unexpected end-of-file: cancer data file
>>>
```

FIGURE 8-31 Unexpected End-of-File Test Case for Function readFiles

We next enter the names of equal length data files to test if the lists returned by readFiles contain the proper data. The results are given in Figure 8-32.

```
Enter file name with smoking data: CDC_Cigarette_Smoking_Data.csv
Enter file name with lung cancer data: CDC_Lung_Cancer_Data.csv

List of smoking data read
[['Alabama',    '23.3'],    ['Alaska',    '24.2'],    ['Arkansas',    '23.7'],
['California',   '14.9'],   ['Colorado',   '17.9'],   ['Connecticut',   '17'],
['Delaware',  '21.7'],  ['Florida',  '21'],  ['Georgia',  '20'],  ['Hawaii',
'17.5'],  ['Idaho',  '16.8'],  ['Illinois',  '20.5'],  ['Indiana',  '24.1'],
['Iowa',  '21.5'],  ['Kansas',  '20'],  ['Kentucky',  '28.6'],  ['Louisiana',
'23.4'], ['Maine', '20.9'], ['Maryland', '17.8'], ['Massachusetts', '17.8'],
['Michigan',   '22.4'],   ['Minnesota',   '18.3'],   ['Mississippi',   '25.1'],
['Missouri',  '23.3'],  ['Montana',  '19'],  ['Nebraska',  '18.6'],  ['Nevada',
'22.2'], ['New Hampshire', '18.7'], ['New Jersey', '18.1'], ['New Mexico',
'20.2'], ['New York', '18.3'], ['North Carolina', '22.1'], ['North Dakota',
'19.6'],   ['Ohio',   '22.5'],   ['Oklahoma',   '25.1'],   ['Oregon',   '18.5'],
['Pennsylvania',  '21.5'],  ['Rhode  Island',  '19.3'],  ['South  Carolina',
'22.3'], ['South Dakota', '20.4'], ['Tennessee', '22.6'], ['Texas', '18.1'],
['Utah',  '9.8'],  ['Vermont',  '18'],  ['Virginia',  '19.3'],  ['Washington',
'17.1'], ['West Virginia', '25.7'], ['Wyoming', '21.6']]

List of cancer data read
[['Alabama',    '75.1'],    ['Alaska',    '69.8'],    ['Arkansas',    '77.7'],
['California',   '51.6'],   ['Colorado',   '48.4'],   ['Connecticut',   '68.3'],
['Delaware',  '83.5'],  ['Florida',  '68.7'],  ['Georgia',  '71'],  ['Hawaii',
'51.7'],   ['Idaho',   '54'],   ['Illinois',   '71.4'],   ['Indiana',   '77.2'],
['Iowa',  '67.1'],  ['Kansas',  '68.4'],  ['Kentucky',  '97.2'],  ['Louisiana',
'77.5'], ['Maine', '80.1'], ['Maryland', '62.9'], ['Massachusetts', '64.5'],
['Michigan',   '72'],   ['Minnesota',   '55.8'],   ['Mississippi',   '77.1'],
['Missouri',  '77.9'],  ['Montana',  '60.8'],  ['Nebraska',  '61.3'],  ['Nevada',
'73.3'], ['New Hampshire', '66.8'], ['New Jersey', '64.5'], ['New Mexico',
'43.9'], ['New York', '63'], ['North Carolina', '75.3'], ['North Dakota',
'53'],   ['Ohio',   '72.8'],   ['Oklahoma',   '80.8'],   ['Oregon',   '64.4'],
['Pennsylvania',   '70'],   ['Rhode  Island',   '69.5'],   ['South  Carolina',
'70.7'], ['South Dakota', '59.2'], ['Tennessee', '82.2'], ['Texas', '62'],
['Utah',  '28'],  ['Vermont',  '79.2'],  ['Virginia',  '65.8'],  ['Washington',
'65.2'], ['West Virginia', '91'], ['Wyoming', '48.9']]
>>>
```

FIGURE 8-32 Proper Data Files Test Case for Function readFiles

Looking at the output, it appears that the data from both files has been properly read and properly constructed in each list.

Development and Unit Testing of Function **calculateCorrelation** The final function of the program to unit test is function calculateCorrelation. An implementation of this function is given in Figure 8-33.

```
import math

def calculateCorrelation(smoking_data, cancer_data):

    """ Calculates and returns the correlation value for the data
        provided in lists smoking_data and cancer_data
    """

    # init
    sum_smoking_vals = sum_cancer_vals = 0
    sum_smoking_sqrd = sum_cancer_sqrd = 0
    sum_products = 0

    # calculate intermediate correlation values
    num_values = len(smoking_data)

    for k in range(0,num_values):
        sum_smoking_vals = sum_smoking_vals + float(smoking_data[k][1])
        sum_cancer_vals = sum_cancer_vals + float(cancer_data[k][1])

        sum_smoking_sqrd = sum_smoking_sqrd +  \
                            float(smoking_data[k][1]) ** 2
        sum_cancer_sqrd = sum_cancer_sqrd +  \
                            float(cancer_data[k][1]) ** 2

        sum_products = sum_products + float(smoking_data[k][1]) *  \
                       float(cancer_data[k][1])

    # calculate and display correlation value
    numer = (num_values * sum_products) - \
            (sum_smoking_vals * sum_cancer_vals)

    denom = math.sqrt(abs( \
        ((num_values * sum_smoking_sqrd) - (sum_smoking_vals ** 2)) * \
        ((num_values * sum_cancer_sqrd) - (sum_cancer_vals ** 2)) \
        ))

    return numer / denom
```

FIGURE 8-33 Implementation of Function calculateCorrelation

The test driver for this function is given in Figure 8-34. For testing the correctness of the calculation performed by function calculateCorrelation, we do not use the data from the data files. Instead, we use data for which we know what the calculated correlation value is. Therefore, three sets of data are hard-coded in the test driver—values in which there is a perfect correlation, another set of values with no correlation, and a final set of values in which there is a perfect negative correlation.

```
1  # TEST DRIVER for Function calculateCorrelation of Smoking/Cancer
2  # Correlation Program
3
4  from module_calculateCorrelation import *
5
6  # calculate perfect positive correlation
7  print('Calculating perfect positive correlation ...')
8  smoking_data = [['A', 10], ['B', 20], ['C', 30], ['D', 40]]
9  cancer_data = [['A', 100], ['B', 200], ['C', 300], ['D', 400]]
10
11 print('Correlation value:', calculateCorrelation(smoking_data, cancer_data))
12
13 # calculate zero correlation
14 print('\nCalculating zero correlation ...')
15 smoking_data = [['A', 10], ['B', 20], ['C', 30], ['D', 40]]
16 cancer_data = [['A', 100], ['B', 0], ['C', 300], ['D', 0]]
17
18 print('Correlation value:', calculateCorrelation(smoking_data,
19 cancer_data))
20
21 # calculate perfect negative correlation
22 print('\nCalculating perfect negative correlation ...')
23 smoking_data = [['A', 10], ['B', 20], ['C', 30], ['D', 40]]
24 cancer_data = [['A', 400], ['B', 300], ['C', 200], ['D', 100]]
25
26 print('Correlation value:', calculateCorrelation(smoking_data, cancer_data))
```

FIGURE 8-34 Test Driver for Function calculateCorrelation

In the first set of lists, the values in list smoking_data (**line 8**) range from 10 to 40 in increments of ten. The values in list cancer_data (**line 9**) range from 100 to 400, in increments of 100. Since the rise in value in each list in proportion to the other, there is a perfect correlation; thus there is, a correlation value of 1. (Note that we simply use the letters ‘A’, ‘B’, etc., for the names of the states since this information is irrelevant for this testing.)

In the second set of lists, half of the values in list cancer_data (**line 15**) rise as the values in list smoking_data rise (**line 16**), and the other half of values decrease as the values in smoking_data rise (each by proportional amounts). Therefore, there is no correlation between these two lists of values; thus, there is a correlation value of 0.

Finally, in the third set of lists, the values in list smoking_data (**line 23**) increase as the values in list cancer_data decrease (**line 24**), each by proportional amounts. Therefore, there is a perfect negative correlation; thus there is a correlation value of 21. The results of the execution of the test driver are given in

Figure 8-35.

At this point, each of the three functions has been successfully unit tested. What follows next is to perform integrating testing by incorporating each of the functions into the main module and testing the program as a whole.

Integration Testing of the Smoking/Cancer Correlation Program

For the purposes of unit testing, we developed each function in its own module. Now that we are ready to perform integration testing, we put all function definitions into one program file with the main module. The complete program is given in Figure 8-36.

```
Calculating perfect positive correlation ...
Correlation value: 1.0

Calculating zero correlation ...
Correlation value: 0.0

Calculating perfect negative correlation ...
Correlation value: -1.0
>>>
```

FIGURE 8-35 Test Results for Function calculateCorrelation

```
Cigarette Use / Lung Cancer Correlation Program
import math

def openFiles():

    """ Prompts the user for the file names to open, opens the files,
        and returns the file objects for each in a tuple of the form
        (smoking_datafile, cancer_datafile).

        Raises an IOError exception if the files are not successfully
        opened after four attempts.
    """

    # init
    smoking_datafile_opened = False
    cancer_datafile_opened = False
    num_attempts = 4

    # prompt for file names and attempt to open files
    while ((not smoking_datafile_opened) or \
           (not cancer_datafile_opened)) \
            and (num_attempts > 0):
        try:
            if not smoking_datafile_opened:
                file_name = input('Enter smoking data file name: ')
                smoking_datafile = open(file_name, 'r')
                smoking_datafile_opened = True

            if not cancer_datafile_opened:
                file_name = input('Enter lung cancer data file name: ')
                cancer_datafile = open(file_name, 'r')
                cancer_datafile_opened = True
        except IOError:
            print('File not found:',file_name + '.','Please reenter\n')
            num_attempts = num_attempts - 1

    # if one of more file not opened, raise IOError exception
    if not smoking_datafile_opened or not cancer_datafile_opened:
        raise IOError('Too many attempts of reading input files')

    # return file objects if successfully opened
    else:
        return (smoking_datafile, cancer_datafile)
```

```
def readFiles(smoking_datafile, cancer_datafile):

    """ Reads the data from the provided file objects smoking_datafile
        and cancer_datafile. Returns a list of the data read from each
        in a tuple of the form (smoking_datafile, cancer_datafile).
    """

    # init
    smoking_data = []
    cancer_data = []
    empty_str = ''

    # read past file headers
    smoking_datafile.readline()
    cancer_datafile.readline()

    # read data files
    eof = False

    while not eof:

        # read line of data from each file
        s_line = smoking_datafile.readline()
        c_line = cancer_datafile.readline()

        # check if at end-of-file of both files
        if s_line == empty_str and c_line == empty_str:
            eof = True

        # check if end of smoking data file only
        elif s_line == empty_str:
            raise IOError('Unexpected end-of-file: smoking data file')

        # check if at end of cancer data file only
        elif c_line == empty_str:
            raise IOError('Unexpected end-of-file: cancer data file')

        # append line of data to each list
        else:
            smoking_data.append(s_line.strip().split(','))
            cancer_data.append(c_line.strip().split(','))

    # return list of data from each file
    return (smoking_data, cancer_data)

def calculateCorrelation(smoking_data, cancer_data):

    """ Calculates and returns the correlation value for the data
        provided in lists smoking_data and cancer_data
    """

    # init
    sum_smoking_vals = sum_cancer_vals = 0
    sum_smoking_sqrd = sum_cancer_sqrd = 0
    sum_products = 0

```

```
    # calculate intermediate correlation values
    num_values = len(smoking_data)

    for k in range(0,num_values):

        sum_smoking_vals = sum_smoking_vals + float(smoking_data[k][1])
        sum_cancer_vals = sum_cancer_vals + float(cancer_data[k][1])

        sum_smoking_sqrd = sum_smoking_sqrd +  \
                                float(smoking_data[k][1]) ** 2
        sum_cancer_sqrd = sum_cancer_sqrd +  \
                                float(cancer_data[k][1]) ** 2

        sum_products = sum_products + float(smoking_data[k][1]) *  \
                       float(cancer_data[k][1])

    # calculate and display correlation value
    numer = (num_values * sum_products) - \
            (sum_smoking_vals * sum_cancer_vals)

    denom = math.sqrt(abs( \
        ((num_values * sum_smoking_sqrd) - (sum_smoking_vals ** 2)) * \
        ((num_values * sum_cancer_sqrd) - (sum_cancer_vals ** 2)) \
        ))

    return numer / denom

# ---- main

# program greeting
print('This program will determine the correlation (-1 to 1) between')
print('data on cigarette smoking and incidences of lung cancer\n')

try:
    # open data files
    smoking_datafile, cancer_datafile = openFiles()

    # read data
    smoking_data, cancer_data = readFiles(smoking_datafile, cancer_datafile)

    # calculate correlation value
    correlation = calculateCorrelation(smoking_data, cancer_data)

    # display correlation value
    print('r_value = ', correlation)
except IOError as e:
    print(str(e))
    print('Program terminated ...')
```

FIGURE 8-36 Cigarette Use/Lung Cancer Correlation Program

We test the program with the same data used for the unit testing of function calculateCorrelation. In this case, however, the data is contained in files to be opened and read by the program. The files and their contents are given below.

```
Smoking Test Data 1
A, 10
B, 20
C, 30
D, 40
```

```
Cancer Test Data 1
A, 100
B, 200
C, 300
D, 400
```

```
Smoking Test Data 2
A, 10
B, 20
C, 30
D, 40
```

```
Cancer Test Data 2
A, 100
B, 0
C, 300
D, 0
```

```
Smoking Test Data 3
A, 10
B, 20
C, 30
D, 40
```

```
Cancer Test Data 3
A, 400
B, 300
C, 200
D, 100
```

The three test results of the program for the data sets above is given below in Figure 8-37. The results are as expected. We get the same results as in the unit testing of
calculateCorrelation. Therefore, all three functions are properly integrated into the program. We next run the program on the actual data files
CDC_Cigaratte_Smoking_Data and
CDC_Lung_Cancer_Data.

```
This program will determine the correlation (-1 to 1) between
data on cigarette smoking and incidences of lung cancer

Enter smoking data file name: Smoking_Test_Data_1.csv
Enter lung cancer data file name: Cancer_Test_Data_1.csv
r_value =  1.0
>>> ================================ RESTART ================================
>>>
This program will determine the correlation (-1 to 1) between
data on cigarette smoking and incidences of lung cancer

Enter smoking data file name: Smoking_Test_Data_2.csv
Enter lung cancer data file name: Cancer_Test_Data_2.csv
r_value =  0.0
>>> ================================ RESTART ================================
>>>
This program will determine the correlation (-1 to 1) between
data on cigarette smoking and incidences of lung cancer

Enter smoking data file name: Smoking_Test_Data_3.csv
Enter lung cancer data file name: Cancer_Test_Data_3.csv
r_value =  -1.0
>>>
```

FIGURE 8-37 Integration Testing Results for Cigarette Use/Lung Cancer Correlation Program

Chapter Summary 331

8.5.5 Determining the Correlation Between Smoking and Lung Cancer The output from the execution of this program on the actual data is given in Figure 8-38.

```
This program will calculate the correlation value (-1 to 1) between
provided data on cigarette smoking and incidences of lung cancer

Enter the names of the two data files, including file extension
(files must contain same number of lines of data)

Enter name of cigarette smoking file: CDC Cigarette Smoking Data.csv
Enter name lung cancer file: CDC Lung Cancer Data.csv
r_value =  0.786159246886
```

FIGURE 8-38 Output of the Cigarette Use/Lung Cancer Correlation Program

In the output, we see a correlation value of approximately 0.79. That indicates a very strong correlation, depicted in Figure 8-39.

FIGURE 8-39 Depiction of Results on a Correlation Scale

So, can we conclude from this result alone that smoking *causes* lung cancer? No. Causation cannot be easily determined even with a perfect correlation. For example, there is a strong correlation between the amount of ice cream sold and the number of drownings that occur in the United States. Does that mean that eating ice cream raises the likelihood of drowning? Common sense tells us no, that the correlation results from them both being summertime activities.

To interpret our results, we have to decide that either (a) smoking causes lung cancer; (b) lung cancer causes the urge to smoke; (c) there exists a third factor simultaneously causing the desire for smoking and lung cancer; or (d) there is no causal relationship. Since a strong correlation is not enough to argue that causation exists, other evidence needs to be considered, such as the lung tissue of those who have died from lung cancer. This, in fact, is what medical research has investigated. Based on the findings, along with the demonstrated correlation, tells us that yes, smoking does cause lung cancer.

CHAPTER SUMMARY General Topics

Text Files vs. Binary Files Opening and Closing Files Reading and Writing Text Files String Processing
Exception Handling
Comma Separated Data Files Python-Specific Programming Topics

Built-in Functions open and close in Python File Object Methods readline and write in

Python
String Methods in Python
Catching and Handling Exceptions in Python Standard Exceptions in Python

CHAPTER EXERCISES

Section 8.1

1. Explain the difference in how numbers are represented in a text file versus how they are represented in a binary file containing numerical values.
2. Explain why binary files do not contain newline characters.

Section 8.2

3. Give an instruction in Python that opens a file named 'datafile.txt' for reading and assigns identifier input_file to the file object created.
4. Give an instruction in Python that opens a file named 'datafile2.txt' for writing and assigns identifier output_file to the file object created.
5. Assume that input_file is a file object for a text file open for reading, and output_file is a file object for a text file open for writing. Explain the contents of the output file after the following code is executed,

```
empty_str 5 "
line 5 input_file.readline()

while line ! 5 empty_str:
output_file.write(line 1 '\n')
line 5 input_file.readline()
```

Section 8.3
6. Give a for loop that counts all the letter characters in string line.
7. For variable month which contains the full name of any given month, give an instruction to display just the first three letters of the month.
8. Give an instruction that displays True if the letter ‘r’ appears in a variable named month, otherwise displays False.
9. Give an instruction for determining how many times the letter ‘r’ appears in a variable named month. **10.** For variables first_name and last_name, give an instruction that displays the person’s name in the form *last name, first name*.
11. Give an instruction that determines if a variable named ss_num contains any non-digit characters, other than a dash.
12. Give an instruction that determines the index of the ‘@’ character in an email address in variable email_ addr.
13. For variable date containing a date of the form 12/14/2012, write a function that produces the same date, but with all slashes characters replaced with dashes.
14. For a variable named err_mesg that contains error messages in the form **

error message **, give an instruction that produces a string containing the error message without the leading and trailing asterisks and blank characters.

Section 8.4
15. Identify the error in the following code,
input_file_opened 5 False while not input_file_opened:

try:
file_name 5 input('Enter file name: ') input_file 5 open(file_name, 'r')

except:
print('Input file not found – please reenter')
PYTHON PROGRAMMING EXERCISES
P1. Write a Python function called reduceWhitespace that is given a line read from a text file and returns the line with all extra whitespace characters between words removed,
‘This line has extra space characters’ → ‘This line has extra space characters’ **P2.** Write a Python function named extractTemp that is given a line read from a text file and displays the one number (integer) found in the string,
‘The high today will be 75 degrees’ → 75
P3. Write a Python function named checkQuotes that is given a line read from a text file and returns True if each quote characters in the line has a matching quote (of the same type), otherwise returns False. ‘Today’s high temperature will be 75 degrees’ →False

P4. Write a Python function named countAllLetters that is given a line read from a text file and returns a list containing every letter in the line and the number of times that each letter appears (with upper/lower case letters counted together),

‘This is a short line’ →[('t', 2), ('h', 2), ('i', 3), ('s', 3), ('a', 1), ('o', 1), ('r', 1), ('l', 1), ('n', 1), ('e', 1)] **P5.** Write a Python function named interleaveChars that is given *two* lines read from a text file, and returns a single string containing the characters of each string interleaved,

‘Hello’, ‘Goodbye’ →'HGeololdobye'
P6. Write a program segment that opens and reads a text file and displays how many lines of text are in the file.

P7. Write a program segment that reads a text file named original_text, and writes every other line, starting with the first line, to a new file named half_text.
P8. Write a program segment that reads a text file named original_text, and displays how many times the letter ‘e’ occurs.

PROGRAM MODIFICATION PROBLEMS

M1. Sparse Text Program: User-Selected Letter Removed
Modify the Sparse Text program in section 8.3.4 so that instead of the letter ‘e’ being removed, the user is prompted for the letter to remove.

M2. Sparse Text Program: Random Removal of Letters
Modify the Sparse Text program in section 8.3.4 so that instead of a particular letter removed, a percentage of the letters are randomly removed based on a percentage entered by the user.

M3. Word Frequency Count Program: Display of Scanned Lines
Modify the Word Frequency Count program in section 8.4.6 so that the text lines being scanned are at the same time displayed on the screen.

M4. Word Frequency Count Program: Counting of a Set of Words
Modify the Word Frequency Count Program so that the user can enter any number of words to be counted within a given text file.

M5. Word Frequency Count Program: Counting of All Words
Modify the Word Frequency Count program so that all the words in a given text file are counted.
M6. Word Frequency Count Program: Outputting Results to a File
Modify the Word Frequency Count program so that the counts of all words in a given text file are output to a file with the same name as the file read, but with the file extension '.wc' (for ‘word count’).
M7. Lung Cancer Correlation Program: Air Pollution and Lung Cancer
Modify the cigarettes and lung cancer correlation program in the Computational Problem Solving section of the chapter to correlate lung cancer with air pollution instead. Use the data from the following ranking of states from highest to lowest amounts of air pollution given below.

Rank	State	Added cancer risk (per 1,000,000)
1.	NEW YORK	1900
2.	NEW JERSEY	1400
3.	DISTRICT OF COLUMBIA	1100
4.	CALIFORNIA	890
5.	MASSACHUSETTS	890
6.	MARYLAND	870
7.	DELAWARE	860
8.	PENNSYLVANIA	860
9.	CONNECTICUT	850
10.	ILLINOIS	800
11.	OHIO	730
12.	INDIANA	720
13.	RHODE ISLAND	670
14.	MINNESOTA	660
15.	GEORGIA	650
16.	MICHIGAN	640
17.	VIRGINIA	620
18.	LOUISIANA	590
19.	WEST VIRGINIA	560
20.	TEXAS	550
21.	WISCONSIN	540
22.	KENTUCKY	540
23.	TENNESSEE	520
24.	OREGON	520
25.	FLORIDA	510
26.	UTAH	500
27.	WASHINGTON	490
28.	NORTH CAROLINA	480
29.	NEW HAMPSHIRE	470
30.	MISSOURI	460
31.	ARIZONA	440
32.	COLORADO	440
33.	ALABAMA	400
34.	SOUTH CAROLINA	390
35.	NEBRASKA	390
36.	KANSAS	360
37.	IOWA	340
38.	NEVADA	340
39.	ARKANSAS	330
40.	OKLAHOMA	330
41.	MISSISSIPPI	320
42.	VERMONT	270
43.	IDAHO	260
44.	MAINE	240
45.	NEW MEXICO	230
46.	NORTH DAKOTA	200
47.	SOUTH DAKOTA	190
48.	MONTANA	180
49.	WYOMING	140

PROGRAM DEVELOPMENT PROBLEMS

D1. Sentence, Word, and Character Count Program
Develop and test a Python program that reads in any given text file and displays the number of lines, words, and total number of characters there are in the file, including spaces and special characters, but not the newline character, '\n'.

D2. Variation on a Sparsity Program
Develop and test a program that reads the text in a given file, and produces a new file in which the *first occurrence only* of the vowel in each word is removed, unless the removal would leave an empty word (for example, for the word “I”). Consider how readable the results are for various sample text.

D3. Message Encryption/Decryption Program
Develop and test a Python program that reads messages contained in a text file, and encodes the messages saved in a new file. For encoding messages, a simple substitution key should be used as shown below,

A	F
B	G
C	L
D	R
E	P
. .	. .

Each letter in the left column is substituted with the corresponding letter in the right column when encoding. Thus, to decode, the letters are substituted the opposite way. Unencrypted message files will be simple text files with file extension .txt. Encrypted message files will have the same file name, but with file extension .enc. For each message encoded, a new substitution key should be randomly generated and saved in a file with the extension '.key'. Your program should also be able to decrypt messages given a specific encoded message and the corresponding key.

D4. Morse Code Encryption/Decryption Program
Develop and test a Python program that allows a user to open a text file containing a simple message using only the (uppercase) letters A . . . Z, and saves a Morse code version of the message, that is, containing only the characters dash (“-”), dot (“.”). In the encoded version, put the encoding of each character on its own line in the text file. Use a blank line to indicate the end of a word, and two blank lines to indicate the end of a sentence. Your program should be able to both convert an English message file into Morse code, and a Morse code file into English. The Morse code for each letter is given below.

A	• –	N	– •
B	– • • •	O	– – –
C	– • – •	P	• – – •
D	– • •	Q	– – • –
E	•	R	• – •
F	• • – •	S	• • •
G	– – •	T	–
H	• • • •	U	• • –
I	• •	V	• • • –
J	• – – –	W	• – –
K	– • –	X	– • • –
L	• – • •	Y	– • – –
M	– –	Z	– – • •

D5. Universal Product Code Check Digit Verification Program
A *check digit* is a digit added to a string of digits that is derived from other digits in the string. Check digits provide a form of redundancy of information, used for determining if any of the digits in the string are incorrect or misread.

The Universal Product Code on almost all purchase items utilizes a bar code to allow for the scanning of items. Below the bar code is the sequence of digits that the bar code encodes, as illustrated below.
03 6 0 0 0 29 1 4 5 2
The last digit of the product code (2) is a check digit computed as follows,
1. Add up all digits in the odd numbered positions (first, third, fifth, etc., starting with the leftmost digit) excluding the last check digit, and multiply the result by 3,

0 1 6 1 0 1 2 1 1 1 5 5 14, 14 * 3 5 42
2. Add up all digits in the even numbered positions (second, fourth, etc.) excluding the last check digit, 3 1 0 1 0 1 9 1 4 5 16
3. Take the sum of the two previous results mod 10,
(42 1 16) mod 10 5 58 mod 10 5 8
4. Subtract the result from 10 to get the checksum digit.
10] 8 5 **2**
Develop and test a Python program that verifies the check digit of Universal Product Codes.

Dictionaries and Sets CHAPTER 9

Since Chapter 4, we have been using linear (sequential) data structures—lists, tuples, and strings— in which elements are accessed by their index value (that is, by location). There also exist data structures in which elements are not accessed by location, but rather by an associated key value, called dictionaries in Python. We look at both dictionaries and sets in Python in this chapter.

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Explain the concept of an associative data structure ♦ Define and use dictionaries in Python
♦ Define and use sets in Python
♦ Write Python programs using dictionaries and sets
♦ Perform unit testing using test stubs (and test drivers)

CHAPTER CONTENTS
Motivation
Fundamental Concepts
9.1 Dictionary Type in Python
9.2 Set Data Type
Computational Problem Solving
9.3 A Food Co-op’s Worker Scheduling Simulation

MOTIVATION

The vast amounts of data being stored today could not be effectively utilized without being organized in some way. Databases do not simply store large

amounts of data. Rather, they provide a complete database management system (DBMS) used in conjunction with stored data to allow the data to be associated and accessed in various ways.

Database management systems provide access to data without needing to know how the data is physically structured, only how it is *logically* organized. Access is provided by a *query language* that allows arbitrarily complex queries to be made of the data. For example, one can construct a query that returns the set of students who have a perfect 4.0 GPA and are graduating at the end of the current semester. Or, one could construct a more complex query that returns the set of students that have a GPA of 3.6 or above, with senior status, female, a computer science major, and has taken at least twelve credits of biology or eight credits of biology and four credits of chemistry.

Some of the concerns of database management systems include: *integrity of data* —that is, data that is accessible and properly organized; *security* in the control of who has what level of access; and management of *concurrency* issues when multiple users are accessing the same data. Data mining (Figure 9-1) is a relatively new field in which database management systems are used in conjunction with methods from statistics and artificial intelligence to extract patterns from very large amounts of data.

Areas	Data Mining Application
Business	Customer Relationship Management (e.g., to find, attract, retain customers)
Genetics	DNA Analysis (e.g., to understand individual variations in human DNA in relation to disease susceptibility)
National Security	Behavioral Patterns (e.g., to identify possible terrorist activities)
Information Retrieval	Music Information Retrieval (to discover musical patterns used in music retrieval)

FIGURE 9-1 Data Mining Applications

FUNDAMENTAL CONCEPTS

9.1 Dictionary Type in Python

In this section we introduce the notion of an *associative data structure*. Elements of indexed linear data structures, such as lists, are ordered—the first element (at index 0), second element (at index 1), and so forth. In contrast, the elements of an associative data structure are unordered, instead accessed by an associated key value. In Python, an associative data structure is provided by the *dictionary type*. We look at the use of dictionaries in Python next.

9.1.1 What Is a Dictionary?

A **dictionary** is a mutable, associative data structure of variable length. The syntax for declaring dictionaries in Python is given below.

daily_temps 5 {'sun': 68.8, 'mon': 70.2, 'tue': 67.2, 'wed': 71.8, 'thur': 73.2, 'fri': 75.6, 'sat': 74.0}

Dictionary daily_temps stores the average temperature for each day of the week, as we did earlier in Chapter 4 using a list. However, in this case, each temperature has associated with it a unique key value ('sun', 'mon', etc.). Strings are often used as key values. The syntax for accessing an element of a dictionary is the same as for accessing elements of sequence types, except that a key value is used within the square brackets instead of an index value: daily_temps['sun']. A comparison of accessing indexed data structures vs. associative data structures is given in Figure 9-2.

FIGURE 9-2 Indexed vs. Associative Data Structure

On the left is an indexed data structure, and on the right an associative data structure. Although the elements of the associative data structure are physically ordered, the ordering is irrelevant to the way that the structure is utilized. *The location that an element is stored in and retrieved from within an associative data structure depends only on its key value*, thus there is no logical first element, second element, and so forth. The specific location that a value is stored is determined by a particular method of converting key values into index values called *hashing*. Example use of dictionary daily_temps is given below.

if daily_temps['sun'] . daily_temps['sat']:
print 'Sunday was the warmer weekend day'
else
if daily_temps['sun'] , daily_temps['sat']: print 'Saturday was the warmer weekend day' else:
print 'Saturday and Sunday were equally warm'

Although strings are often used as key values, any immutable type may be used as well, such as a tuple (shown in Figure 9-3). In this case, the temperature for a specific date is retrieved by,
temps[('Apr', 14, 2001)] → 74.6

FIGURE 9-3 Associative Data Structure Using Tuple Key Values

Note that this key contains both string and integer values. To give an example of when an associative array may be of benefit over an indexed type, we give two versions of a program that displays the recorded average temperature for any day of the week—one using a list, and the other using a dictionary. We first give the list version of the program in Figure 9-4, which allows a user to enter a day of the week and have the average temperature for that day displayed.

```
# Temperature Display Program (List Version)

daily_temps = [68.8, 70.2, 67.2, 71.8, 73.2, 75.6, 74.0]

print('This program will display the average temperature for a given day')
terminate = False

while not terminate:

    day = input("Enter 'sun', 'mon', 'tue', 'wed', 'thur', 'fri', or 'sat': ")

    if day == 'sun':
        dayname = 'Sunday'
        temp = daily_temps[0]
    elif day == 'mon':
        dayname = 'Monday'
        temp = daily_temps[1]
    elif day == 'tue':
        dayname = 'Tuesday'
        temp = daily_temps[2]
    elif day == 'wed':
        dayname = 'Wednesday'
        temp = daily_temps[3]
    elif day == 'thur':
        dayname = 'Thursday'
        temp = daily_temps[4]
    elif day == 'fri':
        dayname = 'Friday'
        temp = daily_temps[5]
    elif day == 'sat':
        dayname = 'Saturday'
        temp = daily_temps[6]

    print('The average temperature for', dayname, 'was', temp, 'degrees\n')

    response = input('Continue with another day? (y/n): ')
    if response == 'n':
        terminate = True
```

FIGURE 9-4 Temperature Display Program (Indexed Array Version)

The program prompts the user for the day of the week (as 'sun', 'mon', 'tue', etc.) to display the average temperature for. The average temperatures are stored in

list daily_ temps. Once read, an if statement (with elif headers) is used to set variable dayname to the full name of the day entered, as well as retrieve the corresponding temperature from list daily_temps. The result is then displayed to the user. An example execution of this program is given below.

This program will display the average temperature for a given day Enter 'sun', 'mon', 'tue', 'wed', 'thur', 'fri', or 'sat': wed The average temperature for Wednesday was 71.8 degrees ...

The program in Figure 9-5 provides the identical functionality, but instead using an associative array instead of a list for storing the average daily temperatures.

```
# Temperature Display Program (Dictionary Version)

daily_temps = {'sun': 68.8, 'mon': 70.2, 'tue': 67.2, 'wed': 71.8,
               'thur': 73.2, 'fri': 75.6, 'sat': 74.0}

daynames = {'sun': 'Sunday', 'mon': 'Monday', 'tue': 'Tuesday',
            'wed': 'Wednesday', 'thur': 'Thursday', 'fri': 'Friday',
            'sat': 'Saturday'}

print('This program will display the average temperature for a given day')
day = input("Enter 'sun', 'mon', 'tue', 'wed', 'thur', 'fri', or 'sat': ")
print('The average temperature for', daynames[day], 'was',
       daily_temps[day], 'degrees')
```

FIGURE 9-5 Temperature Display Program (Dictionary Version)

This version of the program is much more concise and elegant than the previous one. Rather than using a series of conditions in an if-elif statement for checking the day of the week entered by the user, the entered day is directly used for retrieving the corresponding average temperature from dictionary daily_temps. A second associative array is also used in the program for storing the corresponding full day names, accessed by the same set of key values.

In the previous example, the values of the dictionaries are “hard-coded” into the program. Dictionaries, however, may also be created and modified dynamically (that is, during program execution). There are cases where this capability is needed. For example, if a given dictionary contains thousands of elements, too many to hard-code into a program, then the key/value pairs can be read from a file and the dictionary “built” at runtime. Or, it may be that some of the values to be stored are not yet known, and therefore, the dictionary needs to be expanded and updated at execution time (for example, when storing user-selected passwords). The operations related to the dynamic creation and updating of

dictionaries are shown in Figure 9-6.

Operation	Results
`dict()`	Creates a new, empty dictionary
`dict(s)`	Creates a new dictionary with key values and their associated values from sequence `s`, for example, `fruit_prices = dict(fruit_data)` where `fruit_data` is (possibly read from a file): `[['apples', .66],…,['bananas', .49]]`
`len(d)`	Length (num of key/value pairs) of dictionary `d`.
`d[key] = value`	Sets the associated value for `key` to `value`, used to either add a new key/value pair, or replace the value of an existing key/value pair.
`del d[key]`	Remove key and associated value from dictionary `d`.
`key in d`	`True` if key value `key` exists in dictionary `d`, otherwise returns `False`.

FIGURE 9-6 Some Operations for Dynamically Manipulating Dictionaries

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
... fruit_prices 5 {'apples': .66, 'pears': .25,
'peaches': .74, 'bananas': .49}
... fruit_prices['apples']
???
... fruit_prices[0]
???
... veg_data 5 [['corn', .25], ['tomatoes', .49], ['peas', .39]]
... veg_prices 5 dict(veg_data)
... veg_prices
???
... veg_prices['peas']
???
```

A **dictionary** in Python is a mutable, associative data structure of variable length denoted by the use of curly braces.

9.1.2 Let's Apply It—Phone Number Spelling Program

The following Python program (Figure 9-8) generates all the possible spellings of the last four digits of any given phone number. The program utilizes the following programming features: ➤ dictionaries

Example execution of the program is given in Figure 9-7.

The program begins at **line 55**. First, a program welcome is displayed on **lines 55–56**. On **line 59**, variable terminate is assigned False. This variable controls the while loop at **line 61**. It is set to True, which terminates the loop, when the user indicates that they do not wish to enter any more phone numbers (**line 69**).

Within the loop, functions getPhoneNum and displayAllSpellings are called. Function getPhoneNum (**lines 3–32**) reads a phone number from the user of the form 123-456-7890.

Program execution ...

```
This program will generate all possible spellings of the
last four digits of any phone number

Enter phone number (xxx-xxx-xxxx): 410-555-7324
410-555-pdag
410-555-pdah
410-555-pdai
410-555-pdbg
410-555-pdbh
410-555-pdbi
410-555-pdcg
410-555-pdch
410-555-pdci
410-555-peag
410-555-peah
410-555-peai
410-555-pebg
410-555-pebh
410-555-pebi
.
.
Enter another phone number? (y/n): y
Enter phone number (xxx-xxx-xxxx): 410-555-4267
410-555-gamp
410-555-gamq
410-555-gamr
410-555-gams
410-555-ganp
410-555-ganq
410-555-ganr
410-555-gans
410-555-gaop
410-555-gaoq
410-555-gaor
410-555-gaos
.
.
```

FIGURE 9-7 Execution of Phone Number Spelling Program

On **line 8**, variable valid_ph_num is initialized to False, used to control the while loop on **line 12**. Only when a valid phone number has been entered is this variable set to True, which terminates the loop and returns the (valid) entered phone number.

Within the while loop, the entered phone number is input as a string (**line 13**). Two checks are made for the validity of the entered string. First, a check (**line 16**) is made as to whether the string is the wrong length (should be twelve characters long) or if the fourth (index 3) or eighth (index 7) characters are not a dash. If any of these errors are found, an error message is displayed (**line 18**) and another iteration of the while loop is executed. If, however, no error is found at this point, a second check is made that the remaining characters in the string (other than the two dashes) are digit characters. In this case, the string is assumed to contain proper digit characters (that is, valid_ph_ num is set to True on **line 22**), unless found otherwise (by use of string method isdigit on **line 26**). Thus, if a non-digit character is found, valid_ph_num is set to False and the inner while loop terminates, continuing with another execution of the outer while loop (at **line 12**). If, however, no non-digits are found, valid_ph_num remains True, thus terminating the outer while loop and returning the entered phone number.

```
 1 # Phone Number Spelling Program
 2
 3 def getPhoneNum():
 4
 5     """Returns entered phone number in the form 123-456-7890."""
 6
 7     # init
 8     valid_ph_num = False
 9     empty_str = ''
10
11     # prompt for phone number
12     while not valid_ph_num:
13         phone_num = input('Enter phone number (xxx-xxx-xxxx): ')
14
15         # check if valid form
16         if len(phone_num) != 12 or phone_num[3] != '-' or \
17            phone_num[7] != '-':
18             print('INVALID ENTRY - Must be of the form xxx-xxx-xxxx\n')
19         else:
20         # check for non-digits
21             k = 0
22             valid_ph_num = True
23             phone_num_digits = phone_num.replace('-', empty_str)
24
25             while valid_ph_num and k < len(phone_num_digits):
26                 if not phone_num_digits[k].isdigit():
27                    print('* Non-digit:', phone_num_digits[k],'*\n')
28                     valid_ph_num = False
29                 else:
30                     k = k + 1
31
32     return phone_num
33
34 def displayAllSpellings(phone_num):
35
36     """Displays all possible phone numbers with the last four digits
37        replaced with a corresponding letter from the phone keys
38     """
39     translate =   {'0': ('0'), '1':('1'), '2': ('a','b','c'),
40                    '3': ('d','e','f'),'4': ('g','h','i'),
41                    '5': ('j','k','l'), '6': ('m','n','o'),
42                    '7': ('p','q','r','s'),'8': ('t','u','v'),
43                    '9': ('w','x','y','z')}
44
45     # display spellings
46     for let1 in translate[phone_num[8]]:
47         for let2 in translate[phone_num[9]]:
48             for let3 in translate[phone_num[10]]:
49                 for let4 in translate[phone_num[11]]:
50                     print(phone_num[0:8] + let1 + let2 + let3 + let4)
51
```

FIGURE 9-8 Phone Number Spelling Program (*Continued)*

In function displayAllSpellings (**lines 34–50**), we see our first example of the use of deeply nested (for) loops. Because we want to generate all combinations

of each of the four possible letters of the last four digits of the entered phone number, we utilize for loops nested four deep (**line 46**). Each digit ranges over its particular set of letters that appear on phone keys. Dictionary translate is defined for this purpose. In the dictionary, each digit 0–9 is a key value. The value associated with each key is a tuple containing the associated keypad letters for that

```
52 #---- main
53
54 # program welcome
55 print('This program will generate all possible spellings of the')
56 print('last four digits of any phone number\n')
57
58 # get phone number and display spellings
59 terminate = False
60
61 while not terminate:
62
63     phone_num = getPhoneNum()
64     displayAllSpellings(phone_num)
65
66     # continue?
67     response = input('Enter another phone number? (y/n): ')
68     if response == 'n':
69         terminate = True
```

FIGURE 9-8 Phone Number Spelling Program

digit. (Digits '0' and '1' return a string equal to themselves, since these phone digits do not have any letters associated with them.) For example, within the first for statement, dictionary translate contains the list of letters associated with the digit currently in the eighth position (the digit right after the second dash) of the entered phone number. Let’s say that digit is 7. Then the retrieved tuple would be ('p','q','r','s'). Therefore, all combinations of strings starting with these letters will be generated, as shown in Figure 9-9.

```
let1 = 'p','q','r','s'
     let2 = 'd','e','f'
          let3 = 'a','b','c'
               let4 = 'g','h','i'
                    410-555-pdag
                    410-555-pdah
                    410-555-pdai

                    410-555-pdbg
                    410-555-pdbh
                    410-555-pdbi

                    410-555-pdcg
                    410-555-pdch
                    410-555-pdci

                    410-555-peag
                    410-555-peah
                    410-555-peai

                    etc.
```

FIGURE 9-9 Nested for Loops in the Generation of Phone Number Spellings

On **line 67**, the user is prompted if they want to continue with another phone number. If they respond no ('n'), variable terminate is set to True, which terminates the outer while loop, and thus ends the program; otherwise, the program prompts for another number.

Self-Test Questions

1. A dictionary type in Python is an associative data structure that is accessed by a _____ rather than an index value.

2. Associative data structures such as the dictionary type in Python are useful for, **(a)** accessing elements more intuitively than by use of an indexed data structure **(b)** maintaining elements in a particular order

3. Which of the following types can be used as a key in Python dictionaries? **(a)** strings
(b) lists
(c) tuples
(d) numerical values

4. Which of the following is a syntactically correct sequence, s, for dynamically creating a dictionary using dict(s).
(a)s 5[[1: 'one'], [2: 'two'], [3: 'three']]
(b)s 5[[1, 'one'], [2, 'two'], [3, 'three']]
(c)s 5{1:'one', 2:'two', 3:'three'}

5. For dictionary d 5{'apples':0.66,'pears':1.25,'bananas':0.49}, which of the following correctly updates the price of bananas.
(a)d[2] 5 0.52
(b)d[0.49] 5 0.52
(c)d['bananas'] 5 0.52

ANSWERS: 1. key value, 2. (a), 3. (a), (c), (d), 4. (b), 5. (c)

9.2 Set Data Type
9.2.1 The Set Data Type in Python

A **set** is a mutable data type with nonduplicate, unordered values, providing the usual mathematical set operations as shown in Figure 9-10.

Set operator	Set A = {1,2,3}	Set B = {3,4,5,6}	
membership	`1 in A`	True	*True if 1 is a member of set*
add	`A.add(4)`	{1,2,3,4}	*Adds new member to set*
remove	`A.remove(2)`	{1,3}	*Removes member from set*
union	`A | B`	{1,2,3,4,5,6}	*Set of elements in either set A or set B*
intersection	`A & B`	{3}	*Set of elements in both set A and set B*
difference	`A - B`	{1,2}	*Set of elements in set A, but not set B*
symmetric difference	`A ^ B`	{1,2,4,5,6}	*Set of elements in set A or set B, but not both*
size	`len(A)`	3	*Number of elements in set (general sequence operation)*

FIGURE 9-10 Set Operators
One of the most commonly used set operators is the in operator (which we have been already using with sequences) for determining membership,

... fruit 5 {'apple', 'banana', 'pear', 'peach'} ... fruit
{'pear', 'banana', 'peach', 'apple'}

... 'apple' in fruit True

Note that the items in the set are not displayed in the order that they were defined. Sets, like dictionaries, do not maintain a logical ordering. The order that items are stored is determined by Python, and not by the order in which they were provided. Therefore, it is invalid and makes no sense to access an element of a set by index value.

The add and remove methods allow sets to be dynamically altered during program execution, as shown below,

... fruit.add('pineapple')
... fruit
{'pineapple', 'pear', 'banana', 'peach', 'apple'}

To define an initially empty set, or to initialize a set to the values of a particular sequence, the set constructor is used,

... set1 5 set() ... vegs 5 ['peas', 'corn'] ... vowels 5 'aeiou' ... len(set1) ... set(vegs)
... set(vowels) 0 {'corn', 'peas'} {'a', 'i', 'e', 'u', 'o'}

Note that set(), and not empty braces are not used to create an empty set, since that notation is used to create an empty dictionary. Because sets do not have duplicate elements, adding an already existing item to a set results in no change to the set.

Finally, there are two set types in Python—the mutable set type, and the immutable frozenset type. Methods add and remove are not allowed on sets of frozenset type. Thus, all its members are declared when it is defined,

... apple_colors 5 frozenset(['red', 'yellow', 'green'])
As shown, the values of a set of type frozenset must be provided in a single list when defined. (A frozenset type is needed when a set is used as a key value in a given dictionary.) LET’S TRY IT From the Python shell, enter the following and observe the results.

... s 5 {1,2,3} ... s 5 set(['apple', 'banana', 'pear']) ... 1 in s ... s
??? ???

... s.add(4) ... s.add('pineapple')

... s ???

??? ... s 5 frozenset(['apple', 'banana', 'pear']) ... s 5 set('abcde') ...
s.add('pineapple')
... s ... ???
???

A **set** is a mutable data structure with nonduplicate, unordered values, providing the usual set operations. A **frozenset** is an immutable set type.

9.2.2 Let's Apply It—Kitchen Tile Visualization Program

The following Python program (Figure 9-12) allows the user to select a particular kitchen tile size, a primary and secondary tile color, the frequency in which the secondary color is to be placed, and a grout color. It then displays the resulting tile pattern. This program utilizes the following programming features:

➤ sets

Example execution of the program is given in Figure 9-11 (shown as grey tones in this image). Program execution begins on **line 154**. Variable tile_area is assigned to a dictionary
that holds the width and height of the turtle window to be created. Variable grout_color_
selection is assigned to a set containing string values 'white', 'gray', 'brown', and 'black'. These strings are recognized in turtle graphics as color values. The user is prompted to
enter one of these values as the grout color, which is set as the background color of the turtle screen
within function layoutTile. Variable scaling is used to appropriately adjust the size of the
displayed tile. Changing this value will make tiles appear larger or smaller on the screen. Function greeting is called on **line 159** to display an explanation of the program to the
user. On **line 162**, the getTileSelections function is called. This function prompts the user
for the tile width and length, the primary and secondary colors of the tiles, the number of primary
colors that should be displayed for each secondary tile color, and the grout color. These values are

returned in a dictionary of the form,

{'tile_size':{'length':tile_length, 'width':tile_width}, 'primary_color':color1, 'secondary_color':color2, 'tile_skip':skip, 'grout_color':grout}

```
Program Execution ...

This program will display the tile pattern for a selected
pair of tile colors, tile size, and grout color.

The repeat frequency for the secondary tile color can be selected:
if skip = 1, secondary tile color placed every other tile,
if skip = 2, secondary tile color placed every third tile, etc.

Enter tile length (1-4 inches): 4
Enter tile width (1-4 inches): 4
Enter primary tile color (6-digit RGB): 006400
Enter secondary tile color (6-digit RGB): D2B4AC
Enter frequency of secondary tile color (1-10): 2
Enter grout color (white, gray, brown or black): white
```

FIGURE 9-11 Execution of the Kitchen Tile Visualization Program

Thus, for example, for variable selections assigned to this dictionary, selections['tile_ size'] gets the selected tile length and width, and selections['primary_color'] gets the primary tile color. Note that the value associated with key 'tile_size' is itself a dictionary of the form,

{'length':tile_length, 'width':tile_width}

Thus, for variable tile_size assigned to this dictionary, tile_size['length'] returns the user-selected tile length, and tile_size['height'] returns the height.

On **line 165**, the setup method of turtle graphics is called to establish the turtle screen. On **line 168**, the reference to the screen created is assigned to variable window. For this particular program, the normal coordinate system in turtle graphics with coordinate (0,0) at the center of the screen is not the most “natural” coordinate system. Turtle graphics allows the coordinate system to be redefined by specifying the coordinates of the bottom-left corner and the top-right corner of the

```python
# Kitchen Tile Visualization Program

import turtle

def greeting():

    """Displays the program greeting on the screen. """

    print('This program will display the tile pattern for a selected')
    print('pair of tile colors, tile size, and grout color.\n')
    print('The repeat frequency for the secondary tile color can be selected:')
    print('if skip = 1, secondary tile color placed every other tile,')
    print('if skip = 2, secondary tile color placed every third tile, etc.\n')

def getTileSelections(grout_color_options):

    """Returns dictionary of the form,

        {'tile_size':dict, 'color1':str, 'color2':str, 'skip':int, 'grout':str}

        where the value of tile_size is a dictionary of the form
        {'tile_length': value, 'tile_width': value},

        color1 and color2 are the selected primary and secondary color code
        strings in hexadecimal format, skip is an integer indicating the
        number of primary colors that occur for every secondary color, and
        grout contains the selected grout color in hexadecimal format.
    """

    # init
    empty_set = set()
    hex_digits = {'0','1','2','3','4','5','6','7','8','9',
                  'A','B','C','D','E','F','a','b','c','d','e','f'}

    # prompt user for tile size (length and width)
    tile_length = int(input('Enter tile length (1-4 inches): '))
    while tile_length <1 or tile_length > 4:
        tile_length = int(input('INVALID SIZE SELECTED - please re-enter: \n'))

    tile_width = int(input('Enter tile width (1-4 inches): '))
    while tile_width < 1 or tile_width > 4:
        tile_width = int(input('INVALID SIZE SELECTED - please re-enter: \n'))

    # prompt user for primary tile color
    color1 = input('Enter primary tile color (6-digit RGB): ')
    while (len(color1) != 6) or \
          (set(color1) - hex_digits != empty_set):
        color1 = input('INVALID RGB color, please re-enter: ')

    # prompt user for secondary tile color
    color2 = input('Enter secondary tile color (6-digit RGB): ')
    while (len(color2) != 6) or \
          (set(color2) - hex_digits != empty_set):
        color2 = input('INVALID RGB color, please re-enter: ')

    # prompt user for skip pattern for colors
    skip = int(input('Enter frequency of secondary tile color (1-10): '))
    while (skip < 1 or skip > 10):
        skip = int(input('INVALID - Please enter 1,2,3,etc. '))

```

```
61      # prompt user for grout color
62      grout = input('Enter grout color (white, gray, brown or black): ')
63      while (grout not in grout_color_options):
64          grout = input('INVALID grout color selection, please re-enter: ')
65
66      # prepend hash sign to indicate RGB string
67      color1 = '#' + color1
68      color2 = '#' + color2
69
70      # create dictionary of selections
71      selections = {'tile_size':{'length':tile_length, 'width':tile_width},
72                    'primary_color':color1, 'secondary_color':color2,
73                    'tile_skip':skip, 'grout_color':grout}
74
75      return selections
76
77  def layoutTiles(window, selections, tile_area, scaling):
78
79      """Displays the tiles in the window starting with the top and working
80          towards the bottom of the window. This function requires that the
81          coordinate (0,0) be set as the top corner of the window.
82      """
83
84      # set background color (as grout color)
85      window.bgcolor(selections['grout_color'])
86
87      # get selected tile size
88      tile_size = selections['tile_size']
89
90      # get turtle
91      the_turtle = turtle.getturtle()
92
93      # scale size of tiles for display
94      scaled_length = scaling * tile_size['length']
95      scaled_width = scaling * tile_size['width']
96
97      # scale grout spacing
98      tile_spacing =  6
99
100     # create tile shape
101     turtle.register_shape('tileshape',
102                           ((0,0), (0, scaled_length),
103                            (scaled_width, scaled_length), (scaled_width, 0)))
104
105     # set turtle attributes
106     the_turtle.setheading(0)
107     the_turtle.shape('tileshape')
108     the_turtle.hideturtle()
109     the_turtle.penup()
110
111     # place first tile at upper left corner
112     loc_first_tile = (-10, tile_area['height'] + 10)
113     the_turtle.setposition(loc_first_tile)
114
115     # init first tile color and counters
116     first_tile_color = 'primary_color'
117     skip_counter = selections['tile_skip']
118     row_counter = 1
119
```

```
    terminate_layout = False
    while not terminate_layout:

        # check if current row of tiles is before right edge of window
        if the_turtle.xcor()< tile_area['width']:

            # check if need to switch to secondary tile color
            if skip_counter == 0:
                the_turtle.color(selections['secondary_color'])
                skip_counter = selections['tile_skip']
            else:
                the_turtle.color(selections['primary_color'])
                skip_counter = skip_counter - 1

            # place current tile color at current turtle location
            the_turtle.stamp()

            # move turtle to next tile location of current row
            the_turtle.forward(scaled_length + tile_spacing)

        # check if current row of tiles at bottom edge of window
        elif turtle.ycor() + scaled_width > 0:

            the_turtle.setposition(loc_first_tile[0],
                                   loc_first_tile[1]- row_counter *
                                   scaled_width - row_counter * tile_spacing)

            row_counter = row_counter + 1
        else:
            terminate_layout = True

# ---- main

# init
tile_area = {'width': 660, 'height': 440}
grout_color_selection = {'white', 'gray', 'brown', 'black'}
scaling = 20

# program greeting
greeting()

# get tile selection and layout details
selections = getTileSelections(grout_color_selection)

# set window size
turtle.setup(tile_area['width'], tile_area['height'])

# get reference to turtle window
window = turtle.Screen()

# set window title
window.title('Kitchen Tile Visualization Program')

# set coordinate system
window.setworldcoordinates(0, 0, tile_area['width'], tile_area['height'])
window.mode('world')

# layout tiles in window
layoutTiles(window, selections, tile_area, scaling)

# terminate program when window closed
turtle.exitonclick()
```

FIGURE 9-12 Kitchen Tile Visualization Program

screen. This is done by call to method setworldcoordinates on **line 174**. We therefore set the bottom-left corner to be coordinate (0,0), and the top-right corner to be coordinate (tile_ area['width'], tile_area['height']). Thus, for the current tile width of 660 and tile height of 400, the top-right coordinate would be (660,400). This is depicted in Figure 9-13.

FIGURE 9-13 Screen Coordinates for the Kitchen Tile Visualization Program

In order to utilize the new coordinate system, the turtle screen mode must be set to 'world' (on line **175**).

The final steps of the program includes a call (**line 178**) to function layoutTiles. It is passed variable window, holding a reference to the turtle screen created, variable selections, which holds all the user-selected options, variable tile_area, which holds the dimensions of the turtle screen, and variable scaling, which indicates how the tile dimensions in dictionary selections are to be scaled. In the last line of the program, **line 181**, turtle method exitonclick() is called so that the program terminates when the turtle window is closed.

We now look at program functions getTileSelections and layoutTile. Function getTileSelections (**lines 15–75**) begins by setting variable empty_set to an empty set, empty_set 5set(). Recall that empty braces are used to create an empty dictionary, and therefore set() is used to created an empty set. Variable empty_set is used for input error checking as we will see. Variable hex_digits is

assigned to a set containing each of the digits 0–9, as well as the uppercase and lowercase letters A–F. Collectively, these are the symbols used in denoting hexadecimal numbers. This set is also used for input error checking. In **lines 35–64**, the user is prompted to enter all the tile selection options. At **line 36**, the user is prompted for the tile length, limited to a size of four inches. Following, at **line 40**, the user is likewise prompted for the tile width, also limited to four inches. At **line 45**, the user is prompted for the primary tile color. This is the color that will appear most often (or just as often, if the selected pattern is one primary color followed by one secondary color, etc.). Finally, at **line 51**, the user is prompted for the secondary color.

Color values are entered as six hexadecimal digits in the range 000000 (black) to FFFFFF (white). Since color values may contain the letters A–F (a–f), as well as the digits 0–9, any entered color value containing characters other than that is invalid. Example color values are given in Figure 9-14.

white	FFFFFF		DarkSalmon	E9967A		Amethyst	9966CC
ivory	FFFFF0		LightSalmon	FFA07A		DarkViolet	9400D3
Linen	FAF0E6		FireBrick	B22222		DarkOrchid	9932CC
Beige	F5F5DC		DarkRed	8B0000		DarkMagenta	8B008B
LightGray	D3D3D3		Tomato	FF6347		Indigo	4B0082
DarkGray	A9A9A9		OrangeRed	FF4500		CornSilk	FFF8DC
SlateGray	708090		DarkOrange	FF8C00		BlanchedAlmond	FFEBCD
PaleGreen	98FB98		Orange	FFA500		Bisque	FFE4C4
SeaGreen	2E8B57		Gold	FFD700		Wheat	F5DEB3
DarkSlateGray	2F4F4F		Yellow	FFFF00		BurlyWood	DEB887
DarkGreen	006400		LightYellow	FFFFE0		Tan	D2B48C
Olive	808000		LemonChiffon	FFFACD		SandyBrown	F4A460
Teal	008080		Moccasin	FFE4B5		GoldenRod	DAA520
Aquamarine	7FFFD4		PaleGoldenrod	EEE8AA		DarkGoldenrod	B8860B
CadetBlue	5F9EA0		Thistle	D8BFD8		Peru	CD853F
SkyBlue	87CEEB		Plum	DDA0DD		Chocolate	D2691E
CornFlowerBlue	6495ED		MediumPurple	9370DB		SaddleBrown	8B4513
SteelBlue	4682B4		Orchid	DA70D6		Sienna	A0522D
Navy	000080		MediumOrchid	BA55D3		Brown	A52A2A

FIGURE 9-14 Selected Hexadecimal Color Codes

The check for invalid color values occurs on **lines 46** and **52**. First, the color values must be six characters long. Then, the color value strings are converted into a set using set(color1) and set(color2). Thus, for example, for color1 equal to FF3243, set(color1) would produce the following set,

{'3', '2', '4', 'F'}
Then, the set difference operator, 2, is applied to this set, and the set hexdigits defined at the start of the function,
set(color1) 2 hexdigits

Recall that the set difference operator, A – B, returns any items that are in set A,

but not in set B. Since set A is the entered color string and set hexdigits contains all the valid characters of a hexadecimal number, if this difference is not empty (the empty set), then color1 (or color 2) must contain invalid characters.

The final items for which the user is prompted is the skip pattern of the tiles and the grout color. The value of skip indicates how many primary color tiles are placed before a secondary color tile is placed. Thus, for a skip value of 1, the primary and secondary tiles will alternate every other tile. For a skip value of 2, two primary color tiles will be placed for every secondary color tile, and so forth. The grout color entered by the user must be one of the string values in set grout_color_options. Thus, if the entered string is not in this set, the user is prompted to re-enter.

Finally, on lines **67** and **68**, the '#' symbol is prepended to each of the color strings, color1 and color2. This indicates in Python that the string contains a hexadecimal value. At **line 71**, a dictionary is built containing all of the selected values to be returned as the function value (**line 75**).

Function layoutTile, **lines 77–153**, is passed all the information needed to lay out the tiles according to the turtle screen size, the user's selections, and the scaling factor to be used. On **line 85**, the background color of the turtle screen is set to the user-selected grout color. On **line 88**, variable tile_size is set to the value for key 'tile_size' in the selections dictionary. Recall that this value is itself a dictionary. Thus, variable tile_size is holding a dictionary with keys 'length' and 'width'.

On **line 91**, the reference to the turtle is set to variable the_turtle. On **lines 94–95** the dimensions of each tile are scaled by the scaling factor in variable scaling, and assigned to variables scaled_length and scaled_width. Since the user is entering tile size in inches, a fourinch by four-inch tile would be displayed as four pixels by four pixels, which is too small for the purpose of displaying tile patterns. Thus, with a scaling factor of 20, for example, a four-by-four inch tile would be displayed as 80 pixels by 80 pixels. The spacing between tiles (in pixels) is set to 6, assigned to variable tile_spacing on **line 98**.

In order to create the appropriate tile shape as specified by the user, the register_shape method in turtle graphics is used to both describe the shape and provide a shape name. Thus, on **line 101**, register_shape is called with the name 'tileshape' as the first parameter value, followed by a list of coordinate values

defining the desired shape. The coordinate values are specified in terms of the values scaled_length and scaled_width to define the specific size of the tile shape to match the user's specified size.

On **lines 106–109**, the turtle attributes are set for laying out tiles. Since the tiles will be laid out from the top of the screen going left to right, the turtle heading is set to 0 (facing right) and the shape is set to 'tileshape' (on **lines 106–107)**. Since the turtle is used to stamp its image where each tile is to be placed, and not be visible or draw lines on the screen, the turtle is hidden (**line 108**) with its pen set to up (**line 109**).

In order for the tile to appear in the correct location, a ten-pixel adjustment is made. Thus, on **line 112**, variable loc_first_tile is set to the coordinate (210, tile_area['height'] 1 10). The turtle is then positioned at this location (**line 113**) to begin the placement of tiles. Since the primary color tile is always the first tile placed, variable first-tile-color is set to the key value 'primary_color' (of dictionary selections). Variable skip_counter is also set to the skip value stored in the selections dictionary with the key value 'tile_skip'. In order to determine where to begin the placement of tiles for each new row, variable row_counter is initialized to 1 and incremented for each next row of tiles. Thus, knowing the number of each new row allows for the coordinate value of each new row to be calculated (since each row of tiles is the same height).

Lines 120–149 perform the placement of tiles. Variable terminate_layout is initialized to False on **line 120**, which controls the while loop on **line 121**. It is not set to True until the last row of tiles has been completed. The if statement at **line 124** checks whether the x-coordinate value of the turtle’s current location is past the right edge of the screen. If not, then the color of the next tile to be placed is determined based on the current value of variable skip_ counter. This variable is used as a “count-down” counter. That is, it is decremented by one each time another tile has been placed. When skip_counter reaches 0, the next tile is set to the secondary color, and the skip_counter is reset to the value in variable tile_skip to again count down to zero (**lines 127–132**). Once the color of the tile has been determined, the stamp method is called on the turtle to “imprint” its image on the screen (**line 135**). The turtle is then moved forward (to the right) by an amount equal to the tile length plus the amount of tile spacing (**line 138**). Thus, the turtle is positioned for the placement of the next tile, and execution returns to the top of the while loop.

If the condition of the if statement on **line 124** is false, that is, the current tile is past the right edge of the screen, then a check is made on **line 141** if the current tile position is past the bottom of the screen (that is, the_turtle.ycor() is less than zero). If not, then the turtle is repositioned to the beginning of the next row of tiles (**lines 143–145**) making use of variable row_counter, as mentioned. Because a new row of tiles is started, row_counter is incremented by one. Finally, if the conditions at both **line 124** and **line 141** are found false (that is, the current turtle position is at the end of the last row of tiles), then terminate_layout is set to True and thus the while loop terminates, also terminating the function.

Self-Test Questions

1. Indicate all of the following that are syntactically correct for creating a set. **(a)** set([1, 2, 3])
(b)set((1, 2, 3))
(c) {1, 2, 3}

2. For set s containing values 1, 2, 3, and set t containing 3, 4, 5, which of the following are the correct results for each given set operation?
(a) s | t → {3}
(b)s & t → {1, 2, 3, 4, 5}
(c) s 2 t → {1, 2}
(d)s ^ t → {1, 2, 4, 5}

3. For set s containing values 1, 2, 3 and set w of type frozenset containing values 'a','b','c', which of the following are valid set operations?
(a) 'a' in s
(b)'a' in w
(c) len(s) 1 len(w)
(d)s.add(4)
(e) w.add('d')
(f) s | w
(g) s & w
(h)s 2 w

ANSWERS: 1. (a), (b), (c), 2. (c), (d), 3. (a), (b), (c), (d), (f), (g)

COMPUTATIONAL PROBLEM SOLVING

9.3 A Food Co-op’s Worker Scheduling Simulation

In this section, we develop a program for simulating the number of people that

show up for work at a food co-op using two different worker scheduling methods —one in which workers sign up for certain time slots, and the other in which workers show up whenever they want.

9.3.1 The Problem

A food co-op in a university town found an interesting solution to a scheduling problem. The co-op offered two prices for everything— one price for members, and a higher price for nonmembers. To qualify for the lower prices, each member had to volunteer to work at the co-op for a couple hours each week.

The problem was that the co-op needed two people to be working at all times. Members would be asked to sign up for time slots, limited to two workers for each slot. Too often, however, members who had signed up for a given time did not show up, leaving the co-op either completely or partially uncovered.

The co-op eventually devised an effective, albeit daring solution to this problem. They decided to just let members come in to work whenever they wanted to, with no planned scheduling! This unscheduled approach ran the risk of there being times in which the co-op was left without any workers. What was interesting was that they found this approach to be a better solution than the scheduled approach. (We will look at whether this is also better for the individual members or not.)

In this section, we develop a program capable of performing a simulation of both approaches based on assumed probabilities of typical human behavior. We then compare the effectiveness of the scheduled vs. the unscheduled approach both from the co-op’s and the members’ point of view.

9.3.2 Problem Analysis

The computational issue for this problem is to model and simulate the behavior of individuals for assumed probabilities of certain actions. The behaviors in this case are related to fulfilling a commitment to work a given number of hours each week. In one scenario, workers sign up in advance to work certain time slots; in the other scenario, workers show up to work whenever they feel like it.

Besides assumed probabilities of workers showing up for work, there are also assumed probabilities of workers showing up late or leaving early. Since each of these actions is probabilistic, there needs to be a computational means of determining when such actions take place. We use a random number generator for this. For example, if there is an assumed 10% chance that any given person may show up late, a random number between 1 and 10 is generated. If the generated value is 1, the action is assumed to occur; otherwise, the action is assumed not to occur.

We will assume a certain schedule of hours that the co-op is open, given below.

Sunday 12:00 pm–6:00 pm Monday–Thursday 8:00 am–6:00 pm Friday, Saturday 8:00 am–8:00 pm

We also assume that every time slot is two hours long. Thus, for example, there would be three time slots on Sundays, 12:00–2:00 pm, 2:00–4:00 pm, and 4:00–6:00 pm. Based on this, there are 35 time slots in a week. We also assume that the co-op has 75 members.

Because of the different natures of the two scheduling approaches, we assume different probabilities for the behaviors of members. For the scheduled approach,

we assume a probability of 15% that a given member *will not* show up for their time slot. For the unscheduled approach, we assume a probability that any given worker *will* decide to show up for a given time slot to be 5%. Also, we assume that the chance that a scheduled worker will show up late for the start of the time slot is greater than in the unscheduled approach, since in the unscheduled approach workers show up for the time slot that is convenient for them. On the other hand, we assume a greater chance that unscheduled workers will leave fifteen minutes earlier than scheduled workers, since they may feel less committed to working the complete time slot. We assume the probabilities for these behaviors as given in Figure 9-15.

Scheduled Approach Probabilities	Unscheduled Approach Probabilities
Chance of arriving 15 minutes late is 15%.	Chance of arriving 15 minutes late is 5%.
Chance of arriving 30 minutes late is 5%.	Chance of arriving 30 minutes late is 2%.
Chance of arriving 45 minutes late is 2%.	Chance of arriving 45 minutes late is 1%.
Chance of leaving 15 minutes early is 5%	Chance of leaving 15 minutes early is 10%
Chance of leaving 30 minutes early is 3%	Chance of leaving 30 minutes early is 3%
Chance of not showing up at all is 15%	Change of deciding to show up is 5%

FIGURE 9-15 Assumed Probabilities for the Food Co-op Simulation Program

9.3.3 Program Design

Meeting the Program Requirements

The program must simulate the number of workers that show up for each of the time slots that the co-op is open by utilizing assumed probabilities. The co-op requires two workers in the store at all times that it is open. The program must also utilize the probabilities that workers will show up 15, 30, or 45 minutes late for work, or leave 15 or 30 minutes early.

Data Description

The data that needs to be represented in this program includes the number of co-op members (stored as an integer), the two-hour time slots that a worker may work (stored as a tuple), the probabilities of each of the actions that may occur (stored as a dictionary), and the names of the days of the week (stored as a tuple) as given below:

num_members 5 75
time_slots 5 ('8:00am','10:00am','12:00pm','2:00pm','4:00pm','6:00pm') days 5 ('Sunday','Monday','Tuesday','Wednesday','Thursday','Friday',

'Saturday')
sched_probabilities 5 { 'CLate_15':15, 'CLate_30':5, 'CLate_45':2, 'CLEarly_15':5, 'CLEarly_30':3,'CNoshow':15} unsched_probabilities 5 { 'CLate_15':5, 'CLate_30':2, 'CLate_45':1, 'CLEarly_15':10, 'CLEarly_30':3,'CShowup':5}

In dictionary probabilities, CLate_15, CLate_30, and CLate_45 contain the chances that a worker will arrive 15, 30, or 45 minutes late, respectively; CLEarly_15 and CLEarly_30 contain the chances that a worker will leave 15 or 30 minutes early, respectively; CNoshow contains the chance that a scheduled worker will not show up for their time slot; and CShowup is the chance that an unscheduled worker will decide to show up to work.

Algorithmic Approach

The algorithmic approach for this problem relies on the use of randomization to run the simulation. One complete week of co-op staffing is simulated, including the number of workers showing up for each of the 35 time slots in a week, how many show up a given number of minutes late, and how many leave a given number of minutes early. For the assumed probability of each of these actions, a random number will be generated for simulating whether each event has occurred or not.

Overall Program Steps
The overall steps of the program are given in Figure 9-16.

FIGURE 9-16 Overall Steps of Food Co-op Program

Modular Design

The modular design for this program is given in Figure 9-17. Following this design, there are six functions in the program. Function execute ScheduledSimulation determines the time slots for a given day of the week by calling function displayScheduledWorkerHours to probabilistically determine whether a scheduled worker shows up, and if so, if they show up a certain number of minutes late and/or leave a certain number of minutes early. The function, in turn, relies on the use of function eventOccurred, which is provided a probability value, and returns True or False to

FIGURE 9-17 Modular Design of the Food Co-op Simulation Program

simulate whether the event has occurred or not. Function executeUnscheduledSimulation is designed similarly, except that it also needs to determine how many (unscheduled) workers decide to show up at their convenience. Thus, the function relies on function numWorkersShowedUp (which in turn relies on function eventOccurred). We give the specification for each of these functions in Figure 9-18. From this we can begin to implement and test the program.

9.3.4 Program Implementation and Testing

The implementation of the main module of this program is given in Figure 9-19. The main module gives the overall steps of the program. First, all values that determine the model's behaviors are initialized. Variable num_members is assigned the assumed number of co-op members for the simulation, 75 (**line 4**). Tuple time_slots is assigned the start time of all the two-hour time slots that workers may work (**line 5**), and tuple days is assigned the days of the week that the co-op is open (every day of the week) on **line 8**. The dictionary sched_probabilities contains the assumed probabilities for when scheduled workers show up late, leave early, or don't show up at all (**lines 11–12**). The

dictionary unsched_probabilities contains the assumed probabilities for when unscheduled workers show up late, leave early, and decide to show up to work for a given time slot (**lines 14–15**).

On **line 18**, the seed function for the random number generator is called (the required import random statement is provided at the top of the program). When random.seed() is called without an argument, the seed value is taken from the system clock (usually from the lower order digits of the time in the milliseconds range). This ensures that each time the program is run, a different sequence of random numbers most likely will be generated.

The remainder of the program handles the task of prompting the user for the type of simulation to execute (“scheduled” or “unscheduled”). Based on the user’s response, either function executeScheduledSimulation (at **line 36**) or function executeUnscheduledSimulation (at **line 40**) is called. Variable valid_input is initialized to False (at **line 23**), set to True only when a valid response of 1 or 2 is entered for the desired simulation (at **line 43**). Thus,

```
def eventOccurred(chances):
    """For given integer value chances as a percentage value (1-100), returns
       True if randomly generated number in range (1,100) is less than or equal
       to chances, otherwise returns False.
    """
```

```
def numWorkersShowedUp(indiv_chance, num_individuals):
    """For given integer values indiv_chance and num_individuals, returns how
       many times that num_individual calls to event_occurred with argument
       indiv_chance returns True.
    """
```

```
def displayScheduledWorkerHours(workernum, probabilities):
    """For workernum equal to 1 or 2, and the proability values given in dictionary
       probabilities of the form,

         {'CLate_15':<num>, 'CLate_30':<num>, 'CLate_45':<num>,
          'CLEarly_15':<num>, 'CLEarly_30':<num>,'CNoshow':<20>,
          'CShowup':<num>}

       where each <num> is a value between 1-100, displays one of

         worker <worker_num> -- no show --   or
         worker <worker_num> <mins_late> mins late  and/or
         worker <worker_num> left <mins_early> mins early

       where <mins_late> is 15, 30, or 45,  <mins_early> is 15 or 30.
    """
```

```
def displayUnscheduledWorkerHours(workernum, probabilities):
    """For workernum equal to 1 or 2, and the proabilities given in dictionary
       probabilities of the form,

         {'CLate_15':<num>, 'CLate_30':<num>, 'CLate_45':<num>,
          'CLEarly_15':<num>, 'CLEarly_30':<num>,'CNoshow':<20>,
          'CShowup':<num>}

       where each <num> is a value between 1-100, displays one of

         worker <worker_num> -- no show --   or
         worker <worker_num> <mins_late> mins late  and/or
         worker <worker_num> left <mins_early> mins early

       where <mins_late> is 15 or 30,
             <mins_early> is 15 or 30.
    """
```

```
def executeScheduledSimulation(probabilities, days, time_slots):
    """Displays a simulated week of workers in attendance for all time slots,
       based on a scheduled worker schedule.

    """
```

```
def executeUnscheduledSimulation(probabilities, days, time_slots):
    """Displays a simulated week of workers in attendance for all time slots,
        based on an unscheduled worker schedule.
    """
```

FIGURE 9-18 Function Specifications of the Food Co-op Simulation Program

```
# ---- main

# init
num_members = 75
time_slots = ('8:00am', '10:00am', '12:00pm', '2:00pm', '4:00pm',
              '6:00pm')

days = ('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
        'Friday', 'Saturday')

sched_probabilities = {'CLate_15':15, 'CLate_30':5, 'CLate_45':2,
                       'CLEarly_15':5, 'CLEarly_30':3,'CNoshow':15}

unsched_probabilities = {'CLate_15':5, 'CLate_30':2, 'CLate_45':1,
                         'CLEarly_15':10, 'CLEarly_30':3,'CShowup':5}

# seed random number generator with system clock
random.seed()

# get type of simulation
print('Welcome to the Food Co-op Schedule Simulation Program')

valid_input = False
while not valid_input:
    try:
        response = \
            int(input('(1)scheduled, (2)unscheduled simulation? '))

        while (response != 1) and (response != 2):
            print('Invalid Selection\n')
            response = \
                int(input('(1)scheduled, (2)unscheduled simulation? '))

        if response == 1:
            print('<< SCHEDULED WORKER SIMULATION >>\n')
            executeScheduledSimulation(sched_probabilities, days,
                                       time_slots)
        else:
            print('<< UNSCHEDULED WORKER SIMULATION >>\n')
            executeUnscheduledSimulation(unsched_probabilities,
                                         days, time_slots)

        valid_input = True

    except ValueError:
        print('Please enter numerical value 1 or 2\n')
```

FIGURE 9-19 Main Module of the Food Co-op Simulation Program

the while loop continues to execute as long as an invalid response is entered. Because the input value is converted to an integer type,

response 5 int(input('(1) scheduled, (2) unscheduled simulation? '))

a ValueError will be raised (by function input) if the user enters a non-digit

character. Therefore, this line (and the rest of the code) is placed within a try-except block (within **lines 25–45**). Thus, if a non-digit is entered, the raised exception is caught by the except ValueError clause, and the message 'Please enter numerical value 1 or 2' is displayed (at **line 46**). Since valid_input is still False in this case, the while loop performs another iteration, again prompting the user for their selection.

Because a user may enter a numerical value other than 1 or 2, the while loop at **line 29**catches such errors and immediately re-prompts the user. Once the loop terminates, the input is known to be valid. Therefore, based on the selection, either function executeScheduledSimulation or function executeUnscheduleSimulation is called, and valid_input is set to True causing the outer while loop at **line 24** to terminate.

Test Drivers and Test Stubs

In the development of the smoking/lung cancer correlation program of Chapter 8, we utilized unit testing, testing each function separately by developing an appropriate test driver for each. This was followed by integration testing, which tested the complete program with all functions working together.

We use unit testing and integration testing for this program as well. However, in the smoking/ lung cancer program, there were only three functions, each directly called by the main module as given below.

Since these functions did not call any other function of the program, they were simply individually implemented and tested. However, based on the design of the current program, there are functions that call other functions. Thus, these functions cannot be tested without having some version of the other functions to call. This is true for function executeScheduledSimulation for example, shown in Figure 9-20.

FIGURE 9-20 Unit Testing of Function executeScheduledSimulation

As shown in the figure, both a test driver and a test stub are needed for testing the function. A *test driver*, as we saw in Chapter 8, is code developed simply for calling and testing a given function. A *test stub*, on the other hand, is a simple, incomplete implementation of a function for testing functions that call it. For example, a test stub for a function designed to calculate the GPA of a given student might be implemented to simply return an arbitrary floating-point value, *as if* it were a calculated result. Thus, test drivers and test stubs are developed for testing purposes only, and do not become part of the ultimate implementation.

Unit Testing Function **executeScheduledSimulation**
We begin with the unit testing of function executeScheduledSimulation, given in Figure 9-21.

```
def executeScheduledSimulation(probabilities, days, time_slots):

    """Displays a simulated week of workers in attendance for all
       time slots, based on a scheduled worker schedule.
    """

    # for each day of the week
    for day in days:
        print(day)
        print('-------')

        if (day == 'Sunday'):  # sunday?
            current_timeslot = '12:00pm'
            num_timeslots = 3
        elif day in days[1:5]: # mon-thurs?
            current_timeslot = '8:00am'
            num_timeslots = 5
        else:  # friday, saturday
            current_timeslot = '8:00am'
            num_timeslots = 6

        # find loc of current_timeslot in tuple time_slots
        index_val = time_slots.index(current_timeslot)

        # iterate through num_timeslots starting with current time slot
        for time_slot in time_slots[index_val:num_timeslots]:
            print('{0:>7}'.format(time_slot), ' ', end='')

            for k in range(0,2):
                displayScheduledWorkerHours(k+1, probabilities)
                print()
            print()
```

FIGURE 9-21 Implementation of Function executeScheduledSimulation

We develop a test driver that calls function executeScheduledSimulation with values for parameters probabilities, days, and time_slots defined in the main module, given in Figure 9-22.

In addition, since the function depends on the use of function displayScheduledWorkerHours, a test stub is developed for this function for executeScheduled Simulation to call, given in Figure 9-23. All this test stub does is display the value of workernum. The time slots and day of the week are displayed by function execute ScheduledSimulation. Finally, we put function

```
# TEST DRIVER for Function executeScheduledSimulation for Food Co-op Program

from executescheduledsimulation import *

# init required data
sched_probabilities = {'CLate_15':15, 'CLate_30':5, 'CLate_45':2,
                       'CLEarly_15':5, 'CLEarly_30':3,'CNoshow':15}

days = ('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
        'Friday', 'Saturday')

time_slots = ('8:00am', '10:00am', '12:00pm', '2:00pm', '4:00pm',
              '6:00pm', '8:00pm')

# call function
executeScheduledSimulation(sched_probabilities, days, time_slots)
```

FIGURE 9-22 Test Driver for Function executeScheduledSimulation

```
# TEST STUB for Function displayScheduledWorkerHours for Food Co-op Program

def displayScheduledWorkerHours(workernum, probabilities):

    if workernum == 1:
        print('workernum = ', workernum, end=' ')
    else:
        print('workernum = ', workernum)
```

FIGURE 9-23 Test Stub for Function displayScheduledWorkerHours

execute ScheduledSimulation in its own file named executescheduledsimulation. py, as well as the test driver and test stub to be used as modules. Since module executescheduledsimulation needs to call function stub displayScheduledWorkerHours, it contains an import statement of the form from displayscheduledworkerhours import *. Also, since the driver makes a call to function executeScheduledSimulation, it contains an import statement of the form from executescheduledsimulation import *. The result of the testing is given in Figure 9-24.

The output shows the scheduled time slots for each day of the week. However, we see an error for the time slots for Sundays. Only one time slot is displayed (for 8:00 am) but there should be three (8:00 am, 10:00 am, and 12:00 pm). To track down this error, we do some interactive testing in the Python shell. First, we display the value of variable days to make sure it contains each weekday,

... days
('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', ' Saturday')

This is correct. We also check the value of variable time_slots,

```
... time_slots
('8:00am', '10:00am', '12:00pm', '2:00pm', '4:00pm', '6:00pm')
```

```
Sunday
-------
12:00pm  workernum =  1 workernum =  2

Monday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Tuesday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Wednesday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Thursday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Friday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2
 6:00pm  workernum =  1 workernum =  2

Saturday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2
 6:00pm  workernum =  1 workernum =  2
>>>
```

FIGURE 9-24 Unit Testing Results for Function displayScheduledWorkerHours

This is also correct. We then consider the part of function executeScheduledSimulation that determines the time slots for each day,

```
for day in days:
print(day)
print('-------')

if (day 55 'Sunday'): # sunday?
current_timeslot 5 '12:00pm'
num_timeslots 5 3

elif day in days[1:5]: # mon–thurs?
current_timeslot 5 '8:00am'
num_timeslots 5 5

else: # friday, saturday
current_timeslot 5 '8:00am'
num_timeslots 5 6

# find loc of current_timeslot in tuple time_slots
index_val 5 time_slots.index(current_timeslot)
# iterate through num_timeslots starting with current time slot for time_slot in
time_slots[index_val:num_timeslots]: print('{0: .7}'.format(time_slot), ' ', end 5
")
```

We see that variable num_timeslots is set to 3 for Sundays. Since that seems correct, we see what happens in the following code. Variable index is set to the index value of the first occurrence of the first time slot for the day, held by variable current_timeslot. Since the value of current_timeslot should be '12:00pm', we call method index to be sure of the index value returned,

```
... index_val 5 time_slots.index('12:00pm') ... index_val
2
```

This is correct. We therefore focus on the for loop,

```
for time_slot in time_slots[index_val: num_timeslots]: print('{0:
.7}'.format(time_slot), ' ', end 5 ")
```

For all other days, this loop iterates the correct number of times, once for each time slot of each day. We check to see the slice of time_slots that is returned specifically for index_value equal to 2 and num_timeslots equal to 3.

... time_slots[2:3] ('12:00pm',)

Ah, there is a problem! The result should have been ('12:00pm', '2:00pm', '4:00pm'). But why does it work correctly for all the other days of the week? We see that Sunday is the only day that the first time slot does not begin at 8:00 am. Thus, all other slices of list time_slots are of the form,

time_slots[0:k]

whereas for Sunday, it is of the form, time_slots[3:k]

It must come down to how the index values are used in the slice notation. We checked on this and found that we had incorrectly used the slice operation. The first index value is the starting location of the slice, and the second index value is one greater than the last index of the substring. We had written the code as if the second index indicated *how many* elements to include. So the actual code should be,

for time_slot in time_slots[index_val: **index_val** 1 **num_timeslots**]: print('{0: .7}'.format(time_slot), ' ', end 5")

It worked for all the other days of the week because in those cases, index_val was 0, and therefore index_val 1num_timeslots was equal to num_timeslots. So that explains everything! We find a similar error in function executeUnscheduledSimulation, so we make the correction there as well.

We should feel good about ourselves. Not only did we find the error, but we are able to fully explain why this error created the incorrect output. As we have said, making a change that fixes a problem without knowing why the change fixes it is disconcerting and risky. We therefore make this correction to executeScheduledSimulation and again perform unit testing, this time getting the expected output for Sunday (as well as the rest of the days).

```
Sunday
-------
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Monday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Tuesday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Wednesday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Thursday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2
```

```
Friday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2
 6:00pm  workernum =  1 workernum =  2

Saturday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2
 6:00pm  workernum =  1 workernum =  2
>>>
```

We leave as an exercise the unit testing of the remaining program functions.

Integration Testing of the Food Co-op Program
Assuming each function of the program has been unit tested, we next perform integration testing. The complete program is given in Figure 9-25.

The main section of the program is in **lines 213–267**. This part of the program has already been discussed. The displayScheduledWorkerHours, displayUnscheduledWorkerHours, executeScheduledSimulation, and executeUnscheduledSimulation functions contain the bulk of the implementation.

Function executeScheduledSimulation (**lines 135–165**) consists of three nested for loops. The outer loop (**line 142**) iterates through each of the days in the schedule, in this simulation, all seven days. The next inner for loop (**line 160**) iterates through each of the time slots for the day. **Lines 146–154** initialize the first time slot (current_timeslot) as well as the number of time slots (num_timeslots) based on the current value of day. Variable time_slots passed to the function contains all the possible time slots for all the days that the co-op is open. In order to find that first time slot for the current day within this list time_slots, the index method is used (**line 157**), as discussed in the unit testing of this function.

Finally, for each time slot, we iterate over the number of workers for the slot in

the innermost for loop in **lines 162–164**. In this case, the for loop iterates exactly two times since, in the scheduled worker approach, that is the number of workers assigned to each time slot. It is within this innermost loop that function displayScheduledWorkerHours is called (**line 163**). The argument in the function call is adjusted by 1 (k +), because function displayScheduledWorkerHours expects to be passed worker numbers starting at 1.

Function displayScheduledWorkerHours (**lines 38–90**) generates the events that may occur for a given scheduled worker; specifically, whether the worker arrives late and/or leaves early or does not show up at all. Thus, at the start of the function (in **lines 57–58**), variables mins_late and mins_left_early are initialized to zero. Following that is an if statement (**line 60**) determining if the (scheduled) worker does not show up for their assigned time slot. The probability of this occurring is contained in the dictionary probabilities. If the event is found to be true, then mins_late is set to the special value 21, to be used later on **line 80**.

```
# Food Co-op Simulation Program

import random

# Probabilties for Simulation (for values between 1-100)
#
# CLate_xx = chance of being late for work,
# CLEarly_xx = chance of leaving early from work,
# CNoshow = chance that a scheduled worker WILL NOT show up for work
# CShowup = chance that an unscheduled worker WILL show up to work


def eventOccurred(chances):

    """For given integer value chances as a percentage value (1-100),
       returns True if randomly generated number in range (1,100) is
       less than or equal to chances, otherwise returns False.
    """
    if random.randint(1,100) <= chances:
        return True
    else:
        return False

def numWorkersShowedUp(indiv_chance, num_individuals):

    """For given integer values indiv_chance and num_individuals,
       returns how many times that num_individual calls to
       event_occurred with argument indiv_chance returns True.
    """
    numworkers_arriving = 0

    for i in range(num_individuals):
        if eventOccurred(indiv_chance):
            numworkers_arriving = numworkers_arriving + 1

    return numworkers arriving

def displayScheduledWorkerHours(workernum, probabilities):

    """For workernum equal to 1 or 2, and the proability values
       given in dictionary probabilities of the form,

         {'CLate_15':<num>, 'CLate_30':<num>, 'CLate_45':<num>,
          'CLEarly_15':<num>, 'CLEarly_30':<num>,'CNoshow':<num>}


       where each <num> is a value between 1-100, displays one of

         worker <worker_num> -- no show --   or
         worker <worker_num> <mins_late> mins late  and/or
         worker <worker_num> left <mins_early> mins early

       where <mins_late> is 15, 30, or 45,  <mins_early> is 15 or 30.
    """

```

```
    # init
    mins_late = 0
    mins_left_early = 0

    if eventOccurred(probabilities['CNoshow']):
        mins_late = -1
    else:
        if eventOccurred(probabilities['CLate_15']):
            mins_late = 15
        elif eventOccurred(probabilities['CLate_30']):
            mins_late = 30
        elif eventOccurred(probabilities['CLate_45']):
            mins_late = 45

        if eventOccurred(probabilities['CLEarly_15']):
            mins_left_early = 15
        elif eventOccurred(probabilities['CLEarly_30']):
            mins_left_early = 30

    if workernum == 1:
        print('{0:>3}'.format('worker'), str(workernum) + ': ', end='')
    else:
        print('{0:>15}'.format('worker'), str(workernum) + ': ', end='')

    if mins_late == -1:
        print('-- no show --', end='')
    else:
        if mins_late != 0:
            print(mins_late, 'mins late  ', end='')

        if mins_left_early != 0:
            print('left', mins_left_early, 'mins early', end='')

        if (mins_late == 0) and (mins_left_early == 0):
            print('whole time', end='')

def displayUnscheduledWorkerHours(workernum, probabilities):

    """For workernum equal to 1 or 2, and the proability values
       given in dictionary probabilities of the form,

         {'CLate_15':<num>, 'CLate_30':<num>, 'CLate_45':<num>,
          'CLEarly_15':<num>, 'CLEarly_30':<num>,'CShowup':<num>}


       where each <num> is a value between 1-100, displays one of

         worker <worker_num> -- no show --   or
         worker <worker_num> <mins_late> mins late  and/or
         worker <worker_num> left <mins_early> mins early

       where <mins_late> is 15 or 30,
             <mins_early> is 15 or 30.
    """
```

```
    mins_late = 0   # init
    mins_left_early = 0

    if eventOccurred(probabilities['CLate_15']):
        mins_late = 15
    elif eventOccurred(probabilities['CLate_30']):
        mins_late = 30
        elif eventOccurred(probabilities['CLate_45']):
            mins_late = 45

    if eventOccurred(probabilities['CLEarly_15']):
        mins_left_early = 15
    elif eventOccurred(probabilities['CLEarly_30']):
        mins_left_early = 30

    print('{0:>15}'.format('worker'), str(workernum) + ': ', end='')
    if mins_late != 0:
        print(mins_late, 'mins late  ', end='')

    if mins_left_early != 0:
        print('left', mins_left_early, 'mins early', end='')

    if (mins_late == 0) and (mins_left_early == 0):
        print('whole time', end='')

def executeScheduledSimulation(probabilities, days, time_slots):

    """Displays a simulated week of workers in attendance for all
       time slots, based on a scheduled worker schedule.
    """

    # for each day of the week
    for day in days:
        print(day)
        print('-------')

        if (day == 'Sunday'):  # sunday?
            current_timeslot = '12:00pm'
            num_timeslots = 3
        elif day in days[1:5]: # mon-thurs?
            current_timeslot = '8:00am'
            num_timeslots = 5
        else:  # friday, saturday
            current_timeslot = '8:00am'
            num_timeslots = 6

        # find loc of current_timeslot in tuple time_slots
        index_val = time_slots.index(current_timeslot)

        # iterate through num_timeslots starting with current time slot
        for time_slot in time_slots[index_val:index_val + num_timeslots]:
            print('{0:>7}'.format(time_slot), ' ', end='')
            for k in range(0,2):
                displayScheduledWorkerHours(k+1, probabilities)
                print()
            print()
```

FIGURE 9-25 Food Co-op Simulation Program (*Continued*)

If the “no show” event does not occur, the simulation checks events for when the volunteers are 15, 30, or 45 minutes late (**lines 63–68**). Because showing up late and leaving early are independent events, the program also checks for the particular worker leaving 15 or 30 minutes early (**lines 70–73**). The remaining lines of function display_scheduledworkerhours

```
167 def executeUnscheduledSimulation(probabilities, days, time_slots):
168
169     """Displays a simulated week of workers in attendance for all
170        time slots, based on an unscheduled worker schedule.
171     """
172
173     # for each day in the week
174     for day in days:
175         print(day)
176         print('-------')
177
178         if (day == 'Sunday'):   # sunday
179             current_timeslot = '12:00pm'
180             num_timeslots = 3
181         elif day in days[1:5]: # mon-thurs?
182             current_timeslot = '8:00am'
183             num_timeslots = 5
184         else:  # fri, sat
185             current_timeslot = '8:00am'
186             num_timeslots = 6
187
188         # find loc of current_timeslot in tuple time_slots
189         index_val = time_slots.index(current_timeslot)
190
191         # iterate through num_timeslots starting with current time slot
192         for time_slot in time_slots[index_val:index_val + num_timeslots]:
193             numworkers = \
194              numWorkersShowedUp(probabilities['CShowup'], num_members)
195             print('{0:>7}'.format(time_slot), ' ', end='')
196
197             if numworkers == 1:
198                 print(numworkers, 'worker came')
199                 num_stayed = 1
200             elif numworkers == 0 or numworkers == 2:
201                 print(numworkers, 'workers came')
202                 num_stayed = numworkers
203             else:
204                 print(numworkers, 'workers came (' + \
205                 str(numworkers - 2), 'went home)' )
206                 num_stayed = 2
207
208             for k in range(num_stayed): # at most 2 workers stay to work
209                 displayUnscheduledWorkerHours(k+1, probabilities)
210
211             print()
212
```

FIGURE 9-25 Food Co-op Simulation Program (*Continued*)

(**lines 75–90**) involve checking which events occurred and displaying (and properly spacing) the appropriate output.

Functions executeUnscheduledSimulation (**lines 167–211**) and its supporting function displayUnscheduledWorkerHours (**lines 92–133**) are very similar to the corresponding functions executeScheduledSimulation and displayScheduled WorkerHours. One difference is the way that the simulation handles the number of possible workers for a given time slot in function executeUnscheduledSimulation. Because in the unscheduled approach any number of workers can show up, this section indicates how many workers actually stay to work that time

```
213 # ---- main
214
215 # init
216 num_members = 75
217 time_slots = ('8:00am', '10:00am', '12:00pm', '2:00pm', '4:00pm',
218               '6:00pm')
219
220 days = ('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
221         'Friday', 'Saturday')
222
223 sched_probabilities = {'CLate_15':15, 'CLate_30':5, 'CLate_45':2,
224                        'CLEarly_15':5, 'CLEarly_30':3,'CNoshow':15}
225
226 unsched_probabilities = {'CLate_15':5, 'CLate_30':2, 'CLate_45':1,
227                          'CLEarly_15':10, 'CLEarly_30':3,'CShowup':5}
228
229 # seed random number generator with system clock
230 random.seed()
231
232 # get type of simulation
233 print('Welcome to the Food Co-op Schedule Simulation Program')
234
235 valid_input = False
236 while not valid_input:
237     try:
238         response = int(input('(1)scheduled, (2)unscheduled simulation? '))
239
240         while (response != 1) and (response != 2):
241             print('Invalid Selection\n')
242             response = \
243                 int(input('(1)scheduled, (2)unscheduled simulation? '))
244
245         if response == 1:
246             print('<< SCHEDULED WORKER SIMULATION >>\n')
247             executeScheduledSimulation(sched_probabilities,
248                                        days, time_slots)
249         else:
250             print('<< UNSCHEDULED WORKER SIMULATION >>\n')
251             executeUnscheduledSimulation(unsched_probabilities,
252                                          days, time_slots)
253
254         valid_input = True
255
256     except ValueError:
257         print('Please enter numerical value 1 or 2\n')
```

FIGURE 9-25 Food Co-op Simulation Program

slot—no more than two—and how many go home. Another minor difference is the way that displayUnscheduledWorkerHours is designed. This is due to the fact that a “no show” only applies to scheduled workers, and not to unscheduled workers since unscheduled workers never sign up for a time slot to work.

Function event_occurred (**lines 13–22**) is a simple function that returns a Boolean (True/False) value used to simulate whether a given event has occurred or not. The function is passed an integer value between 1 and 100 representing the probability of the event (where 100 represents certainty). It then makes a call to the random number generator to randomly generate an integer in the same range of 1 to 100 (in **line 19**). If the number is less than or equal to the value passed in parameter chance, then the event is assumed to have occurred, and the value True is returned. Otherwise, the event is assumed *not* to have occurred, and thus returns False. This function is what drives the simulation.

Finally , function numworkersShowedUp (**lines 24–36**) is a supporting function, called from **line 194** of function executeUnscheduledSimulation. In the simulation of the unscheduled approach, any number of members may show up for work for any given time slot. To simulate this, function numworkershowedup is designed to be given the chances that an individual member may show up (in parameter indiv_chance), as well as the total number of members in the co-op (in parameter num_individuals).

9.3.5 Analyzing a Scheduled vs. Unscheduled Co-op Worker Approach We now look at the simulation results for worker coverage at the food co-op, one using a scheduled approach (Figure 9-26), and the other using an unscheduled approach (Figure 9-27).

We can compare these two simulation runs. Since each run is based on assumed probabilities of events, the results are only as accurate as the probability estimates. Also, multiple simulation runs would provide a more accurate picture of likely events. Comparative results are given in Figure 9-28.

So with these caveats in mind , what can we conclude from this simulation? We first look at the reason that the food co-op chose to try an unscheduled approach —that too often, time slots were left partially or completely uncovered. From our simulation of a scheduled approach, we do see three times in which time slots

are completely uncovered, and fourteen times in which there is only partial coverage (that is, only one person showing up). In the simulation of the unscheduled approach, there were no time slots left uncovered, and only three time slots in which there was partial coverage. This, therefore, coincides with the co-op's experience that there is better coverage in an unscheduled approach.

We next focus on the number of times a worker was 15 or 30 minutes late. In the scheduled approach, members were 15 minutes late six times, 30 minutes late two times, and 45 minutes late one time. For the unscheduled approach, members were 15 minutes late six times, 30 minutes late three times, and never 45 minutes late. Thus, there is no appreciable difference in the amount of time workers arrive late based on the simulation. When we look at the number of times workers left early, we see that in the scheduled approach, workers left 15 minutes early three times and 30 minutes early four times. In the unscheduled approach, workers left 15 minutes early four times, and 30 minutes early only once. Thus, there is also no appreciable difference in the amount of times workers leave early, except that scheduled workers were more likely to leave 30 minutes early than unscheduled workers (based on our assumed probabilities of human behavior).

Overall, the results indicate that an unscheduled approach does improve the problem of uncovered time slots. This, therefore, improves the functioning of the co-op. However, is there a hidden cost to this? A downside of the unscheduled approach is that there are times when more than the

```
Welcome to the Food Co-op Schedule Simulation
Program
(1)scheduled, (2)unscheduled simulation? 1
<< SCHEDULED WORKER SIMULATION >>

Sunday
-------
12:00pm  worker 1: whole time
         worker 2: whole time

 2:00pm  worker 1: whole time
         worker 2: -- no show --

 4:00pm  worker 1: whole time
         worker 2: -- no show --

Monday
-------
 8:00am  worker 1: -- no show --
         worker 2: whole time

10:00am  worker 1: whole time
         worker 2: whole time

12:00pm  worker 1: -- no show --
         worker 2: -- no show --

 2:00pm  worker 1: left 30 mins early
         worker 2: whole time

 4:00pm  worker 1: whole time
         worker 2: whole time

Tuesday
-------
 8:00am  worker 1: -- no show --
         worker 2: -- no show --

10:00am  worker 1: whole time
         worker 2: whole time

12:00pm  worker 1: whole time
         worker 2: 15 mins late

 2:00pm  worker 1: whole time
         worker 2: -- no show --

 4:00pm  worker 1: whole time
         worker 2: -- no show --

Wednesday
-------
 8:00am  worker 1: whole time
         worker 2: whole time

10:00am  worker 1: 30 mins late
         worker 2: 30 mins late  left 15 mins early

12:00pm  worker 1: 45 mins late  left 30 mins early
         worker 2: whole time

 2:00pm  worker 1: left 15 mins early
         worker 2: 30 mins late

 4:00pm  worker 1: -- no show --
         worker 2: whole time

Thursday
-------
 8:00am  worker 1: whole time
         worker 2: -- no show --

10:00am  worker 1: -- no show --
         worker 2: whole time

12:00pm  worker 1: whole time
         worker 2: 15 mins late

 2:00pm  worker 1: whole time
         worker 2: whole time

 4:00pm  worker 1: whole time
         worker 2: 15 mins late

Friday
-------
 8:00am  worker 1: -- no show --
         worker 2: whole time

10:00am  worker 1: 15 mins late
         worker 2: -- no show --

12:00pm  worker 1: whole time
         worker 2: -- no show --

 2:00pm  worker 1: whole time
         worker 2: left 30 mins early

 4:00pm  worker 1: 15 mins late
         worker 2: whole time

 6:00pm  worker 1: whole time
         worker 2: whole time

Saturday
-------
 8:00am  worker 1: whole time
         worker 2: -- no show --

10:00am  worker 1: whole time
         worker 2: left 30 mins early

12:00pm  worker 1: 15 mins late
         worker 2: whole time

 2:00pm  worker 1: left 15 mins early
         worker 2: -- no show --

 4:00pm  worker 1: -- no show --
         worker 2: -- no show --

 6:00pm  worker 1: whole time
         worker 2: -- no show --
```

FIGURE 9-26 Scheduled Worker Simulation

```
Welcome to the Food Co-op Schedule Simulation
Program
(1)scheduled, (2)unscheduled simulation? 2
<< UNSCHEDULED WORKER SIMULATION >>

Sunday
-------
12:00pm  7 workers came (5 went home)
         worker 1: 15 mins late
         worker 2: whole time

 2:00pm  1 worker came
         worker 1: whole time

 4:00pm  5 workers came (3 went home)
         worker 1: whole time
         worker 2: whole time

Monday
-------
 8:00am  7 workers came (5 went home)
         worker 1: whole time
         worker 2: whole time

10:00am  4 workers came (2 went home)
         worker 1: whole time
         worker 2: whole time

12:00pm  4 workers came (2 went home)
         worker 1: whole time
         worker 2: whole time

 2:00pm  4 workers came (2 went home)
         worker 1: left 15 mins early
         worker 2: left 30 mins early

 4:00pm  4 workers came (2 went home)
         worker 1: 15 mins late
         worker 2: whole time

Tuesday
-------
 8:00am  4 workers came (2 went home)
         worker 1: whole time
         worker 2: 15 mins late

10:00am  5 workers came (3 went home)
         worker 1: 15 mins late
         worker 2: whole time

12:00pm  3 workers came (1 went home)
         worker 1: whole time
         worker 2: whole time

 2:00pm  4 workers came (2 went home)
         worker 1: 15 mins late
         worker 2: whole time

 4:00pm  1 worker came
         worker 1: whole time

Wednesday
-------
 8:00am  4 workers came (2 went home)
         worker 1: left 15 mins early
         worker 2: whole time

10:00am  3 workers came (1 went home)
         worker 1: whole time
         worker 2: left 15 mins early

12:00pm  2 workers came
         worker 1: whole time
         worker 2: left 15 mins early

 2:00pm  6 workers came (4 went home)
         worker 1: whole time
         worker 2: whole time

 4:00pm  6 workers came (4 went home)
         worker 1: whole time
         worker 2: whole time

Thursday
-------
 8:00am  6 workers came (4 went home)
         worker 1: whole time
         worker 2: whole time

10:00am  3 workers came (1 went home)
         worker 1: whole time
         worker 2: whole time

12:00pm  6 workers came (4 went home)
         worker 1: whole time
         worker 2: 30 mins late

 2:00pm  6 workers came (4 went home)
         worker 1: whole time
         worker 2: whole time

 4:00pm  5 workers came (3 went home)
         worker 1: whole time
         worker 2: whole time

Friday
-------
 8:00am  4 workers came (2 went home)
         worker 1: whole time
         worker 2: whole time

10:00am  3 workers came (1 went home)
         worker 1: whole time
         worker 2: whole time

12:00pm  3 workers came (1 went home)
         worker 1: whole time
         worker 2: whole time

 2:00pm  4 workers came (2 went home)
         worker 1: whole time
         worker 2: 30 mins late
```

FIGURE 9-27 Unscheduled Worker Simulation (*Continued*)

```
4:00pm  2 workers came
         worker 1: whole time
         worker 2: whole time

 6:00pm  3 workers came (1 went home)
         worker 1: whole time
         worker 2: whole time

Saturday
-------
 8:00am  6 workers came (4 went home)
         worker 1: whole time
         worker 2: whole time

10:00am  3 workers came (1 went home)
         worker 1: 15 mins late
         worker 2: whole time

12:00pm  9 workers came (7 went home)
         worker 1: whole time
         worker 2: whole time

 2:00pm  1 worker came
         worker 1: whole time

 4:00pm  2 workers came
         worker 1: whole time
         worker 2: 30 mins late

 6:00pm  4 workers came (2 went home)
         worker 1: whole time
         worker 2: whole time
```

FIGURE 9-27 Unscheduled Worker Simulation

two workers show up. Thus, there are expected to be times when members who show up to work are not needed, and therefore need to return another time. In our simulation, this happened 77 times, which depending on other factors (for example, how long it takes to get to the co-op, how far out of the way is it from other locations that members would typically go to, etc.) is an additional burden on individual members.

Thus, it is understandable from the co-op's point of view that an unscheduled approach to worker "scheduling" is a better approach. However, *for individual members it adds an extra burden of time, and thus it may not be considered an improvement from their perspective.*

Number of Time Slots that:	Scheduled Approach	Unscheduled Approach
No One Showed Up	3	0
Only One Person Showed Up	14	3
Number of Times that Someone was 15 Minutes Late	6	6
Number of Times that Someone was 30 minutes Late	3	3
Number of Times that Someone was 45 Minutes Late	1	0
Number of Times that Someone Left 15 Minutes Early	3	4
Number of Times that Someone Left 30 Minutes Early	4	1
Number of People Turned Away Because had Enough Workers (unscheduled only)	-	77

FIGURE 9-28 Results of Simulation Runs of Scheduled vs. Unscheduled Approach

CHAPTER SUMMARY General Topics Python-Specific Programming Topics

Associative Data Structures Set Operations
Hexadecimal Numbers Computer Simulation
Test Drivers/Test Stubs Dictionaries in Python
Set and Frozenset Types in Python

CHAPTER EXERCISES Section 9.1

1. Indicate whether an indexed data structure, an associative data structure, or a set type would be most appropriate for each of the following.
(a) The number of inches of rain for each day of the year in a given locality, used for computing the average yearly rainfall.

(b) The number of inches of rain, only for the days when there was rainfall in a given locality, used to retrieve the amount of rain for any given day as quickly as possible.
(c) Faculty members that belong to various committees, in which each faculty

member on the university senate must also be a member of at least one college committee, but may not also be a member of another university-level committee.
2. Create a dictionary named password_lookup that contains usernames as keys (as string types), and
passwords as associated string values. Make up data for five dictionary entries.
3. Give a program segment that creates an initially empty dictionary named password_lookup, prompting one-by-one for usernames and passwords (until a username of 'z' is read) entering each into the
dictionary.
4. Create a dictionary named password_hint that contains email addresses as keys, and associated values that contain both a user's "password security question," and the answer to the question. Make up data
for five dictionary entries.
5. Create a dictionary named member_table that contains users' email addresses as keys, and their current password as the values. Write a function that generates a temporary new password for a given user
and updates it in the table.

Section 9.2
6. Declare a set named vowels containing the strings 'a','e','i','o', and 'u'. Give a program segment that prompts the user for any English word, and displays how many vowels it contains.

7. Give a program segment that prompts the user for two English words, and displays which letters the two words have in common.
8. Give a program segment that prompts the user for two English words, and displays which letters are in the first word but not in the second.
9. Give a program segment that prompts the user for two English words, and displays which letters of the alphabet are in neither of the two words.
10. Give a program segment that prompts the user for two English words, and displays which letters are in either the first word or the second word, but not in both words.
11. Give a program segment that prompts the user for two English words, entered in no particular order, and determines if all the letters of first word are contained within the second.

PYTHON PROGRAMMING EXERCISES

P1. Write a Python function called addDailyTemp that is given a (possibly

empty) dictionary meant to hold the average daily temperature for each day of the week, the day, and the day's average temperature. The function should add the temperature to the dictionary only if does not already contain a temperature for that day. The function should return the resulting dictionary, whether or not it is updated.

P2. Write a Python function named moderateDays that is given a dictionary containing the average daily temperature for each day of a week, and returns a list of the days in which the average temperature was between 70 and 79 degrees.

P3. Write a Python function named getDailyTemps that prompts the user for the average temperature for each day of the week, and returns a dictionary containing the entered information.

P4. Write a Python function named getWeekendAvgTemp that is passed a dictionary of daily temperatures, and returns the average temperature over the weekend for the weekly temperatures given.

P5. Write a Python function named addVegetable that is passed a (possible empty) set of vegetable names, and raises a ValueError exception if the given vegetable is already in the set, otherwise, the new vegetable should be added and the new set returned.

P6. Write a Python function named numVowels that is passed a string containing letters, each of which may be in either uppercase or lowercase, and returns a tuple containing the number of vowels and the number of consonants the string contains.

PROGRAM MODIFICATION PROBLEMS

M1. Word Frequency Count Program: Making Use of Sets

Modify the Word Frequency Count Program in Chapter 8 making use of the set type where possible.

M2. Phone Number Spelling Program: Modifying for Australia

In Australia, phone numbers are of the form 0x-xxxx-xxxx. Modify the Phone Number Spelling program so that it prints out Australian phone numbers with spellings for the last four digits.

M3. Kitchen Tile Visualization Program: Color Name Entry

Modify the Kitchen Tile Visualization Program so that, instead of requiring the user to enter hexadecimal RGB color codes, they are given a subset of twelve color names from which to select (from the colors listed in Figure 9-11).

M4. Kitchen Tile Visualization Program: Randomly Generated Patterns
Modify the Kitchen Tile Visualization Program so that the user can enter up to three different tile colors, as well as the grout color. The program should then display a pattern of tiles that is randomly generated using the three colors specified.
M5. Kitchen Tile Visualization Program: Automatic Complementary Colors
The complementary color of a given color is the "opposite" color (directly opposite on the color wheel). Designers often use complementary colors because of their vibrant contrast. Modify the Kitchen Tile visualization program so that the user enters one tile color, with the complementary color automatically

generated as the secondary tile color. To calculate the complementary color of a color, each of the three color values is replaced by its arithmetic complement to 255,

FFFFFF (white) → 255 255 255 → 255-255 255-255 255-255 → 000000 (black)
00FF00 (green) → 0 255 0 → 255-0 255-255 255-0 → 255 0 255 → FF00FF (magenta)
87CEEB (sky blue) → 135 206 235 → 255-135 255-206 255-235 → 120 49 20 → 783114 (brownish red)

M6. Kitchen Tile Visualization Program: Muted and Tinted Colors
Sometimes two colors are too vibrant together. Colors can be muted, or made less vibrant and "toned down," by adding some of their complementary color. Modify the Kitchen Tile Visualization Program so that the user can mute either or both of the two displayed colors to varying degrees. (See problem M5 about complementary colors.)

M7. Food Co-op Worker Schedule Simulation Program: Adjusting the Model
The program developed for the food co-op simulation allows for easy adjustment of some of the parameters in the model. This includes the assumed number of co-op members, as well as the various assumed probabilities for members' behaviors. Since the number of members in the co-op only affects the unscheduled simulation results, run the unscheduled simulation for various number of co-op members and
(a) Plot a graph using the x-axis for the number of co-op members, and the y-axis for the number of

members turned away when showing up to work over the period of one week.
(b) Determine the optimal number of co-op members so that each time slot is covered, but with the least
number of members showing up to work turned away.
(c) Adjust the probabilities for the assumed behavior of co-op members to what you consider to be a
more accurate reflection of peoples' behavior and describe the results.
M8. Food Co-op Worker Schedule Simulation Program: Multiple Simulations
Modify the Food Co-op Worker Schedule Simulation program so that the user can select the number of multiple, sequential simulations. The program should output the average results over all the simulation runs as shown below:

♦ average number of workers that showed up to work for any given time slot ♦ average number of workers that worked the complete time slot
♦ average number of workers that were 15, 30, and 45 minutes late
♦ average number of workers that left 15 or 30 minutes early
♦ average number of workers turned away over the week because the co-op had enough

workers (for unscheduled approach)
PROGRAM DEVELOPMENT PROBLEMS

D1. Reverse Phone Spelling Program
Develop and test a program that allows the user to enter a spelled phone number for the last four digits (for example, 410-555-book) and generates the phone number that produces that spelling.

D2. Color Encoding Conversion Program
Develop and test a program that allows the user to enter six-digit hexadecimal RGB color codes and converts them to base 10. (You are *not* to use the '#' symbol for denoting hexadecimal values in Python in this program.) In this format, the first two hexadecimal digits represents the amount of red, the second two the amount of green, and the last two the amount of blue. Following is the hexadecimal value for the color "tomato,"

Hexadecimal numbers have place values that are powers of 16, in which the letters A–F have the values 10–15, respectively. Thus, FF in hexadecimal is equal to 15 * 16 1 15 * 1 5 240 1 15 5 255. Likewise, 63 in hexadecimal is equal to 6 * 16 1 3 * 1 5 96 1 3 5 99.

D3. Color Code Recording Program
Develop and test a program that allows the user to store color codes that they would like to save for future reference. The program should be designed so that they can enter a color code in either hexadecimal or decimal format (see problem D2). Whichever color code format is entered, it should be stored in both hexadecimal and decimal form. For example, if the user enters F0A514, it should be stored as both F0A514 and (240, 165, 20), and vice versa. The program should allow an annotation to be added for each color (for example, "Used this shade of green on my website"). The user should also be able to delete unwanted color entries.

Object-Oriented Programming CHAPTER 10

In Chapter 6, we learned what objects are and how they are used. We found that all values in Python are objects. Classes in object-oriented programming define objects. Thus, a class is a "cookie cutter" for creating any number of objects of that type. As with functions, there are predefined classes in Python, as well as the ability of programmers to define their own. In this chapter, we see how to define and use classes in Python.

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to ♦ Explain the fundamental concepts of object-oriented programming ♦ Explain the

concept of a class
♦ Define encapsulation, inheritance, and polymorphism ♦ Explain the use of subclasses as a subtype
♦ Explain the purpose of UML
♦ Explain the relationships of a UML class diagram
♦ Write simple UML class diagrams
♦ Define and use classes in Python
♦ Explain and use special methods in Python
♦ Effectively use inheritance and polymorphism in Python

CHAPTER CONTENTS
Motivation
Fundamental Concepts
10.1 What Is Object-Oriented Programming?
10.2 Encapsulation
10.3 Inheritance
10.4 Polymorphism
10.5 Object-Oriented Design Using UML
Computational Problem Solving

10.6 Vehicle Rental Agency Program

MOTIVATION

Classification can be described as the act of grouping entities into various categories that have something in common. In Biology, organisms are placed in a taxonomy based on their individual traits, for example. Libraries group books using various classification systems based on subject matter.

Most classification systems contain subcategories (and subcategories of subcategories, etc.), resulting in a hierarchy of types as shown in Figure 10.1. For example, chimpanzees are species in the Hominidae family, which is in the order Primate, which is in the mammal class, which is in the Animal Kingdom. The African elephant, on the other hand, is a species in the Elephantidae family, which is in the order Proboscidea, which is in the same class as chimpanzees, the Mammal class.

By this taxonomy, therefore, chimpanzees and African elephants have the same traits identified in the Mammal class, as well as traits of the Animal Kingdom.

Their differences, on the other hand, are identified in the Primate and Proboscidea orders, as well as the traits within the Hominidae and Elephantidae families, and the chimpanzee and African elephant species. In this chapter, we see how the organization of class type and subtypes are utilized in object-oriented programming.

FIGURE 10-1 Taxonomy of Chimpanzees and African Elephants
FUNDAMENTAL CONCEPTS
10.1 What Is Object-Oriented Programming?

We have discussed and have been using objects in our programs. The use of objects in itself, however, does not constitute object-oriented programming. For that, we first need to introduce the concept of a class, which we discuss next.

10.1 What Is Object-Oriented Programming? 385
10.1.1 What Is a Class?

A **class**specifies the set of instance variables and methods that are "bundled together" for defining a type of object. A class, therefore, is a "cookie cutter" that can be used to make as many object instances of that type object as needed. For example, strings in Python are object instances of the built-in String class, depicted in Figure 10-2.

FIGURE 10-2 Object Instances of String Class
One method of the String class is isdigit. Thus, every string object has this method. The specific object whose isdigit method is called determines the specific string that it is applied to, name.isdigit() →False city_state.isdigit() → False address.isdigit() →False zip_code.isdigit() → True We next look at the three fundamental features of object-oriented programming.
A **class** specifies the set of instance variables and methods that are "bundled together" for defining a type of object.
10.1.2 Three Fundamental Features of Object-Oriented Programming

Object-oriented programming languages, such as Python, provide three

fundamental features that support object-oriented programming—*encapsulation*, *inheritance*, and *polymorphism*. These support a paradigm shift in software development from the focus on variables and passing of variables to functions in procedural programming, to the focus on objects and *message passing* between them.

Message passing occurs when a method of one object calls a method of another, as depicted in Figure 10-3. For example, if Object B were a list and B1 a sorting method, then a call to B1 is a message (or request) for it to become sorted. A message to one object can result in the propagation of messages among many other objects in order to accomplish a request. In the next sections we discuss these three features of object-oriented programming, as well as introduce a means of specifying an object-oriented design using UML.

FIGURE 10-3 Message Passing in Object-Oriented Programming

Three fundamental features supporting the design of object-oriented programs are referred to as *encapsulation*, *inheritance*, and *polymorphism*.

10.2 Encapsulation

In this section we develop a Fraction class for demonstrating the notion of encapsulation. (Note that the Fraction class developed here is unrelated to the Fraction class of the Python Standard Library.)

10.2.1 What Is Encapsulation?

Encapsulation is a means of bundling together instance variables and methods to form a given type (class). Selected members of a class can be made inaccessible (“hidden”) from its clients, referred to as *information hiding*. Information hiding is a form of abstraction. This is an important capability that object-oriented programming languages provide. As an example, we give a depiction of a Fraction object in Figure 10-4.

FIGURE 10-4 Fraction Object Access

The Fraction object shows private instance variables __numerator and __denominator, and four public methods. Private members of a class begin with two underscore characters, and cannot be directly accessed. For example, trying to access the instance variables of Fraction object frac1 is invalid,

frac1.__numerator 5 4 NOT ALLOWED frac1.__denominator 56 NOT ALLOWED
Public members of a class, on the other hand, are directly accessible. For example, the following are valid method calls,

frac1.getNumerator() ALLOWED frac1.getDenominator() ALLOWED frac1.setNumerator(4) ALLOWED frac1.setDenominator(6) ALLOWED

These methods are referred to as **getters** and **setters** since their purpose is to get (return) and set (assign) private instance variables of a class. Restricting access to instance variables via getter and setter methods allows the methods to control what values are assigned (such as not allowing an assignment of 0 to the denominator), and how they are represented when retrieved. Thus, the instance variables of a class are generally made private, and the methods of the class generally made public.

Encapsulation is a means of bundling together instance variables and methods to form a given type, as well as a way of restricting access to certain class

members.
10.2.2 Defining Classes in Python In this section, we develop a Python Fraction class.
Defining a Fraction Class
The first stage in the development of a Fraction class is given in Figure 10-5.

The class keyword is used to define classes, much as def is used for defining functions. All lines following the class declaration line are indented. Instance variables are initialized in the __init__ special method. (We discuss special methods in the following.) Being private, instance variables __numerator and __denominator are not meant to be directly accessed.

```
... frac1.__numerator
AttributeError: 'Fraction' object has no attribute '__numerator'
```

In actuality, however, private members *are* accessible if written as follows:

```
... frac1._Fraction__numerator
```

```
class Fraction(object):
                                                        Special Methods

    def __init__(self, numerator, denominator):
        """Inits Fraction with values numerator and denominator."""

        self.__numerator = numerator
        self.__denominator = denominator
        self.reduce()
                                                Getter and Setter Methods
    def getNumerator(self):
        """Returns the numerator of a Fraction."""

        return self.__numerator

    def getDenominator(self):
        """Returns the denominator of a Fraction."""

        return self.__denominator

    def setNumerator(self, value):
        """Sets the numerator of a Fraction to the provided value."""

        self.__numerator = value

    def setDenominator(self, value):
        """Sets the denominator of a Fraction to the provided value.

          Raises a ValueError exception if a value of zero provided.
        """

        if value == 0:
            raise ValueError('Divide by Zero Error')

        self.__denominator = value
```

FIGURE 10-5 Initial Fraction Class

To understand this, all private class members are automatically renamed to begin with a single underscore character followed by the class name. Such renaming of identifiers is called *name mangling*. Unless the variable or method is accessed with its complete (mangled) name, it will not be found. Name mangling prevents unintentional access of private members of a class, while still allowing access when needed.

The methods of a class, as we have seen, are essentially functions meant to operate on the instance variables of the class. In Python, functions serving as a method must have an extra first parameter, by convention named self. This parameter contains a reference to the object instance to which the method

belongs. When a method accesses any other member of the same class, the member name must be preceded by 'self' (self.__numerator). Getter and setter methods are also defined. Note that setDenominator raises an exception when passed a value of 0 to ensure that Fraction objects cannot be set to an invalid value. We further discuss special methods in Python next.
LET'S TRY IT

Enter and execute the following Python class. Then enter the given instructions within the Python shell and observe the results.

class SomeClass(object): ... obj 5 SomeClass() def __init__(self):... obj.__n
self.__n 5 0???
self.n2 5 0 ... obj._SomeClass__n

???
... obj.n2
???

The class keyword is used to define a class in Python. Class members beginning with two underscore characters are intended to be private members of a class. This is effectively accomplished in Python by the use of *name mangling*.

Special Methods in Python

Method names that begin and end with two underscore characters are called *special methods* in Python. Special methods are automatically called. For example, the __init__ method of the Fraction class developed is automatically called whenever a new Fraction object is created,

frac1 5Fraction(1,2) – creates new fraction with value 1/2 frac2 5Fraction(6,8) – creates new fraction with value 6/8

The values in parentheses are arguments to the __init__ method to initialize a new Fraction object to a specific value. Note that although there are three parameters defined (self, numerator, denominator), the first is always implied. Therefore, only the remaining arguments (numerator and denominator) are explicitly provided when creating a Fraction object.

Two other special methods of Python are __str__ and __repr__. These methods are used for representing the value of an object as a string. The __str__ method

is called when an object is displayed using print (and when the str conversion function is used.) The __repr__ function is called when the value of an object is displayed in the Python shell (when interactively using Python). This is demonstrated below.

class DemoStrRepr(): ... s 5 DemoStrRepr() def __repr__(self): ... print(s) return '__repr__ called' __str__ called def __str__(self): ... s return '__str__ called' __repr__ called

The difference in these special methods is that __str__ is for producing a string representation of an object's value that is most readable (for humans), and __repr__ is for producing a string representation that Python can evaluate. If special method __str__ is not implemented, then special method __repr__ is used in its place. An implementation of __repr__ for the Fraction class is given below.

def __repr__(self):
return str(self.__numerator) 1 '/' 1 str(self.__denominator) This, therefore, will display Fraction values as we normally write them:

... frac1 5 Fraction(3,4) ... print('Value of frac1 is', frac1) ... frac1 Value of frac1 is 3/4
3/4

We give special method __repr__ of the Fraction class (to also serve as the implementation of special method __str__) in Figure 10-6.

```
class Fraction(object):

                                                    Additional Special Methods
    def __repr__(self):
        """Returns Fraction value as x/y."""

        return str(self.__numerator) + '/' + str(self.__denominator)
```

FIGURE 10-6 Additional Special Method repr
We will look at more Python special methods in the next section.
LET'S TRY IT
Enter and save the following class definition in a Python file and execute it. Then enter the given instructions in the Python shell and observe the results.
class XYcoord(object):

def __init__(self, x, y): self.__x 5 x
self.__y 5 y
... coord 5 XYcoord(5, 2) ... print(coord)
???

... str(coord) ???
def __repr__(self):
return '(' 1 str(self.__x) 1 ',' \

1 str(self.__y) 1 ')' ... coord ???

Special methods in Python have names that begin and end with two underscore characters, and are automatically called in Python.
Adding Arithmetic Operators to the Fraction Class
Special methods used to provide arithmetic operators for class types are shown in Figure 10-7.

Operator	Example Use	Special Method
- (negation)	-frac1	__neg__
+ (addition)	frac1 + frac2	__add__
- (subtraction)	frac1 - frac2	__sub__
* (multiplication)	frac1 * frac2	__mul__

FIGURE 10-7 Arithmetic Operator Special Methods

The expression frac1 1frac2, for example, evaluates to the returned value of the __add__ method in the class, where the left operand (frac1) is the object on which the method call is made: frac1.__add__(frac2). Arithmetic methods for the Fraction class are given in Figure 10-8.

Arithmetic Operator Special Methods

```
    def __neg__(self):
        """Returns a new Fraction equal to the negation of self."""

        return Fraction(-self.__numerator, self.__denominator)

    def __add__(self, rfraction):
        """Returns a new reduced Fraction equal to self + rfraction."""

        numer = self.__numerator * rfraction.getDenominator() + \
                rfraction.getNumerator() * self.__denominator

        denom = self.__denominator * rfraction.getDenominator()

        resultFrac = Fraction(numer, denom)
        return resultFrac.reduce()

    def __sub__(self, rfraction):
        """Returns a new reduced Fraction equal to self - rfraction."""

        return self + (-rfraction)

    def __mul__(self, rfraction):
        """Returns a new reduced Fraction equal to self * rfraction."""

        numer = self.__numerator * rfraction.getNumerator()
        denom = self.__denominator * rfraction.getDenominator()

        resultFrac = Fraction(numer, denom)
        resultFrac.reduce()

        return resultFrac
```

FIGURE 10-8 Arithmetic Operators of the Fraction Class

Special method __add__ is implemented to add the numerator and denominator of each fraction based on a common denominator as shown below,

2/4 1 1/5 → (2 * 5)/(4 * 5) 1 (1 * 4)/(5 * 4) → 10/20 4/20
→ 14/20
→ 7/10 (after call to method reduce)

Note that a reduce method of the class (not given) is called on the resulting fraction to return the result in simplest form. Thus, rather than returning 14/20, the value 7/10 is returned.

The __sub__ special method has a very simple and elegant implementation. It simply adds the first fraction to the negation of the second fraction, relying on the implementation of the __add__ and __neg__ methods. Finally, the __mul__

special method multiplies the numerators of each of the fractions, as well as the denominators of each, building a new Fraction object from the results, and reducing the new fraction to simplest form by call to method reduce. (We omit the special method for the division operator here.)

Just as with the other arithmetic operators, a new (negated) Fraction object is returned by the __neg__ method, rather than the original fraction object being altered. To negate a fraction, reassignment would be used instead,

frac1 5 2frac1
LET’S TRY IT Enter and save the following class definition in a Python file, and execute. Then enter the given instructions in the Python shell and observe the results. class XYcoord(object):

def __init__(self, x, y): self.__x 5 x
self.__y 5 y

def __repr__(self) return '(' 1 str(self.__x) 1 ',' \ ???

1 str(self.__y) 1 ')' def __add__(self, rCoord):
new_x 5 self.__x 1 rCoord.__x
new_y 5 self.__y 1 rCoord.__y
return XYCoord(new_x, new_y)
... coord_1 5 XYcoord(4,2) ... coord_2 5 XYCoord(6,10) ... coord_1 1 coord_2
???

... coord 5 coord_1 1 coord_2 ... print(coord)
Adding Relational Operators to the Fraction Class
We have yet to add the capability of applying relational operators to Fraction objects. The set of relational operator special methods that can be implemented is shown in Figure 10-9.

Operator	Example Use	Special Method
< (less than)	frac1 < frac2	__lt__
<= (less than or equal to)	frac1 <= frac2	__le__
== (equal to)	frac1 == frac2	__eq__
!= (not equal to)	frac1 != frac2	__ne__
> (greater than)	frac1 > frac2	__gt__
>= (greater than or equal to)	frac1 >= frac2	__ge__

FIGURE 10-9 Relational Operator Special Methods

The special methods for providing the relational operators of the Fraction class are given in Figure 10-10.

Relational Operators

```
def __eq__(self, rfraction):
    """Returns True if self arithmetically equal to rfraction.
       Otherwise, returns False.
    """

    temp_frac1 = self.copy()
    temp_frac2 = rfraction.copy()
    temp_frac1.reduce()
    temp_frac2.reduce()

    return temp_frac1.getNumerator() == temp_frac2.getNumerator() and \
       temp_frac1.getDenominator() == temp_frac2.getDenominator()
```

```
def __neq__(self, rfraction):
    """Returns True if Fraction not arithmetically equal to rfraction.
       Otherwise, returns False.
    """

    return not self.__eq__(rfraction)
```

```
def __lt__(self, rfraction):
    """Returns True if self less than rfraction."""

    if self.getDenominator() == rfraction.getDenominator():
        return self.getNumerator() < rfraction.getNumerator()
    else:
        temp_frac1 = self.copy()
        temp_frac2 = rfraction.copy()

        saved_denom = temp_frac1.getDenominator()
        temp_frac1.__adjust(temp_frac2.getDenominator())
        temp_frac2.__adjust(saved_denom)

        return temp_frac1.getNumerator() < temp_frac2.getNumerator()
```

```
def __le__(self, rfraction):
    """Returns True if self less than or equal to rfraction."""

    return not (rfraction < self)
```

```
def __gt__(self, rfraction):
    """Returns True if self greater than rfraction."""

    return not(self <= rfraction)
```

```
def __ge__(self, rfraction):
    """Returns True if self greater than or equal to rfraction."""

    return not(self < rfraction)
```

FIGURE 10-10 Relational Operators of the Fraction Class

Each of the special methods for the relational operators is implemented. For example, frac1 , frac2 is determined by call to method __lt__ on the first object, frac1, with the second object, frac2, passed as an argument,

frac1.__lt__(frac2)

In order to compare two fractions, they must have common denominators. Therefore, method __ lt__ first checks if the denominators are equal. If so, then the result of self.__getNumerator() , rfraction.__getNumerator()is returned. Otherwise, since we do not want the two fractions to be altered as a result of the comparison, a copy of each is made, assigned to temp_frac1 and temp_frac2. In order to convert them common denominators, the numerator and denominator of each is multiplied by the denominator of the other. This is accomplished by call to private method __adjust. Then, the Boolean result temp_frac1.__getNumerator(),temp_frac2.__ getNumerator() is returned.

Most other relational operators are grounded in the implementation of the less than special method, __lt__. The implementation of special method __le__ (less than or equal to) is based on the fact that a ,5 b is the same as not (b , a). Special method __neq__ (not equal to) is simply implemented as not (a 5 b). Finally, special method __gt__ (greater than) is implemented as not a ,5 b, and special method __ge__ (greater than or equal to) is implemented as not (a , b). Finally, the implementation of methods copy, reduce, and __adjust are left as an exercise at the end of the chapter.

The Fraction type developed represents the power of abstraction, as implemented through the use of encapsulation. The Fraction class contains two integer instance variables and an associated set of methods. That is what it “really” is. Through the use of information hiding, however, the client is provided an abstract view, which, for all intents and purposes, *is*a Fraction type. The Fraction class can be defined within its own Python file, and thus serve as a module that can be easily imported and used by any program.

There exist special methods for the arithmetic and relational operators in Python that can be implemented to determine how these operators are evaluated for a given object type.

10.2.3 Let’s Apply It—A Recipe Conversion Program

The following Python program (Figure 10-12) will convert the measured amount of ingredients of recipes, based on a provided conversion factor, to vary the number of servings. The program utilizes the following programming features:

➤ programmer-defined class

Example execution of the program is given in Figure 10-11. The program makes use of the fraction class module developed
in section 10.2.2, imported on **line 3**. Since there is only one item to
be imported (the class), the choice of using import Fraction vs.
from Fraction import * only affects whether Fraction objects
are created as frac1 5Fraction.Fraction(1,2) for the
first form of import, or frac1 5Fraction(1,2) for the second

Chocolate Chip Cookies Recipe

4 1/2 cups all-purpose flour
2 teaspoons baking soda
2 cups butter, softened
1 1/2 cups packed brown sugar
1/2 cup white sugar
2 packages instant vanilla pudding mix
4 eggs
2 teaspoons vanilla extract
4 cups semisweet chocolate chips
2 cups chopped walnuts

```
This program will convert a given recipe to a different
quantity based on a specified conversion factor. Enter a
factor of 1/2 to halve, 2 to double, 3 to triple, etc.

Enter file name: ChocChipCookiies.txt
File Open Error

Enter file name: ChocChipCookies.txt
Enter the conversion factor: 2
Converted recipe in file:  conv_ChocChipCookies.txt
```

```
Chocolate Chip Cookies Recipe
9 cups all-purpose flour
4 teaspoons baking soda
4 cups butter, softened
3 cups packed brown sugar
1 cup white sugar
4 packages instant vanilla pudding mix
8 eggs
4 teaspoons vanilla extract
8 cups semisweet chocolate chips
4 cups chopped walnuts
```

```
Enter file name: ChocChipCookies.txt
Enter the conversion factor: 1/2
Converted recipe in file:  conv_ChocChipCookies.txt
```

```
Chocolate Chip Cookies Recipe
9/4 cups all-purpose flour
1 teaspoons baking soda
1 cups butter, softened
3/4 cups packed brown sugar
1/4 cup white sugar
1 packages instant vanilla pudding mix
2 eggs
1 teaspoons vanilla extract
2 cups semisweet chocolate chips
1 cups chopped walnuts
```

FIGURE 10-11 Execution of the Recipe Conversion Program

form. We therefore choose the from-import form of import. Note that the methods of a class are called the same way regardless of the form of import used to import the class.

The main section of the program is in **lines 106–143**. The program welcome is provided on **lines 109–111**. The rest of the code is encompassed within a try block for catching any IOError exceptions. There is one instance of an IOError exception that is raised by the program (in addition to those raised by the Python standard functions) in function getFile (**lines 5–28**).

The getFile function prompts for a file name to open, returning both the file name and associated file object as a tuple. If, after three attempts the file fails to open successfully, an IOError exception is raised containing the error message 'Exceeded number of open file attempts' (**line 26**). Thus, a try block is used to catch each IOError exception raised by the open function. When such an exception is caught, variable num_attempts is incremented (**line 22**) in the corresponding exception handler (**lines 21–23**). When an input file is successfully opened, the loop terminates and file_name and input_file are returned as a tuple.

Back in the main module of the program, the user is prompted for the conversion factor (**line 119**). Since all calculations in the program are executed as Fraction types, the conversion factor is scanned (read) by call to function scanAsFraction (**lines 45–84**). If a single integer value (read as a string) is entered, for example '2', then scanAsFraction returns the Fraction value 2/1. If a single fraction value is entered, such as '2/4', then the Fraction value (in reduced form) 1/2 is returned. If an integer and fraction are entered, such as '1 1/2', then a Fraction equal to the sum of both is returned, 3/2.

Function scanAsFraction returns, as a Fraction value, the total value of the initial part of the parameter string passed.
4 1/2 cups all-purpose flour

```python
# Recipe Conversion Program

from fraction import *

def getFile():

    """Returns as a tuple the file name entered by the user and the
       open file object. If the file exceeds three attempts of opening
       successfully, an IOError exception is raised.
    """

    file_name = input('Enter file name: ')
    input_file_opened = False
    num_attempts = 1

    while not input_file_opened and num_attempts < 3:
        try:
            input_file = open(file_name, 'r')
            input_file_opened = True
        except IOError:
            print('File Open Error\n')
            num_attempts = num_attempts + 1
            file_name = input('Enter file name: ')

    if num_attempts == 3:
        raise IOError('Exceeded number of file open attempts')

    return (file_name, input_file)

def removeMeasure(line):

    """Returns provided line with any initial digits and fractions
       (and any surrounding blanks) removed.
    """

    k = 0
    blank_char = ' '

    while k < len(line) and (line[k].isdigit() or \
                             line[k] in ('/', blank_char)):
        k = k + 1

    return line[k:len(line)]

def scanAsFraction(line):

    """Scans all digits, including fractions, and returns as a
       Fraction object. For example, '1/2' would return as Fraction
       value 1/2, '2' would return as Fraction 2/1, and '2 1/2' would
       return as Fraction value 3/2.
    """

```

```
    completed_scan = False
    value_as_frac = Fraction(0,1)

    while not completed_scan:
        k = 0
        while k < len(line) and line[k].isdigit():
            k = k + 1

        numerator = int(line[0:k])

        if k < len(line) and line[k] == '/':
            k = k + 1
            start = k
            while k < len(line) and line[k].isdigit():
                k = k + 1

            denominator = int(line[start:k])
        else:
            denominator = 1

        value_as_frac = value_as_frac + Fraction(numerator,
                                                 denominator)

        if k == len(line):
            completed_scan = True
        else:
            line = line[k:len(line)].strip()

            if not line[0].isdigit():
                completed_scan = True

    return value_as_frac

def convertLine(line, factor):

    """If line begins with a digit, returns line with the value
       multiplied by factor. Otherwise, returns line unaltered
       (e.g., for a factor of 2, '1/4 cup' returns as '1/2 cup’.)
    """

    if line[0].isdigit():
        blank_char = ' '
        frac_meas = scanAsFraction(line) * factor

        if frac_meas.getDenominator() == 1:
            frac_meas = frac_meas.getNumerator()

        conv_line = str(frac_meas) + blank_char + removeMeasure(line)
    else:
        conv_line = line

    return conv_line
```

```
106 # ---- main
107
108 # display welcome
109 print('This program will convert a given recipe to a different')
110 print('quantity based on a specified conversion factor. Enter a')
111 print('factor of 1/2 to halve, 2 to double, 3 to triple, etc.\n')
112
113 try:
114
115     # get file name and open file
116     file_name, input_file = getFile()
117
118     # get conversion factor
119     conv_factor = input('Enter the conversion factor: ')
120     conv_factor = scanAsFraction(conv_factor)
121
122     # open output file named 'conv_' + file_name
123     output_file_name = 'conv_' + file_name
124     output_file = open(output_file_name, 'w')
125
126     # convert recipe
127     empty_str = ''
128     recipe_line = input_file.readline()
129
130     while recipe_line != empty_str:
131         recipe_line = convertLine(recipe_line, conv_factor)
132         output_file.write(recipe_line)
133         recipe_line = input_file.readline()
134
135     # close files
136     input_file.close()
137     output_file.close()
138
139     # display completion message to user
140     print('Converted recipe in file: ', output_file_name)
141
142 except IOError as err_mesg:  # catch file open error
143     print(err_mesg)
```

FIGURE 10-12 Recipe Conversion Program

First, variable completed_scan is initialized to False (**line 53**) and set to True when the initial digit (and '/') characters have been scanned. Then variable value_as_frac (**line 54**) is initialized to the Fraction value 0/1, so that it can be used to accumulate values (**line 73**) when both an integer and a fractional value are found, as shown. Within the while loop, index variable k is initialized to 0 (**line 57**). Following that, k is incremented to scan past each character in line that is a digit character (as long as k is less than the length of the line). This places k at the index location of the first non-digit. Then, the range of characters scanned so far, line[0:k], is converted to an integer type and assigned to numerator. The next character is then scanned for the ‘/’ character (indicating that a fraction

notation exists), as long as the end of line has not been reached (that is, k ,len(line)). If the slash character is found, then there is a denominator to go with the numerator value just scanned. Thus, the following characters are scanned until a non-digit is found (**lines 66–67**).

The current location of k is first stored in variable start (**line 65**) so that the beginning of the new substring of digits can be scanned. At that point, denominator is set to the integer value of the digits from index start to k21 (line[start:k]. Then, on **line 73**, variable value_ as_frac is set to the current value of value_as_frac plus the newly created Fraction object with the current values of numerator and denominator. If a slash character is not found, then denominator is set to one (**line 71**) so that the integer value scanned, for example 2, is returned as 2/1.

On **line 76** a check is made to determine if the end of the line has been reached. (This would not normally be true of a recipe line, but would be True when scanning a conversion value.) If the end has been reached, then completed_scan is set to True, which terminates the main while loop, causing the final return statement to return the scanned value. If, however, the end of the line has not been reached, the next character is checked to see if it is a digit. If it is, then the while loop at **line 56** iterates again to scan the following value, which is expected to be a fractional value. If the next character is not a digit, then completed_scan is set to True (**line 82**), causing the function to return the (non-fractional) value scanned.

In **lines 123–124** of the main module, output_file_name is assigned to the name of the input file name and prepended with the string 'conv_', and file object output_file is created (where the converted recipe is written). Following that, the process of converting each line of the recipe file begins. First, empty_str is initialized and the first line of the recipe file is read as recipe_line (**lines 127–128**). The while loop at **line 130** then converts the current recipe line by a call to convertLine. Function convertLine (**lines 86–104**) first checks that the first character of the line is a digit. If not, then the line is assigned unaltered to conv_line on **line 102** (since there is no initial numerical value to convert). If a digit is found, then blank_char is initialized (**line 94**), and frac_meas is set to the Fraction value returned by scanAsFunction (for the given conversion factor). If the denominator of frac_meas is found to be 1 (e.g., 2/1), then frac_meas is set to the value of the numerator (2), otherwise it is left as is. Variable line is set to the remaining part of the line by call to removeMeasure (**line 100**), and

conv_line is set to the string representation of frac_meas concatenated with a blank and the remaining part of the original line, and returned. Finally, function removeMeasure (**lines 30–43**) scans past the initial digit, blank, and slash characters, returning the remaining part of the line.

Self-Test Questions

1. Encapsulation bundles together instance variables and methods of a class, and allows certain members of the class to be made inaccessible by use of keyword private in Python. (TRUE/ FALSE)

2. Private members of a class are not meant to be directly accessed by methods of other classes. (TRUE/FALSE)
3. Methods of a class that provide access to privates members of the class are called _____ and _____.

4. Which of the following are special methods in Python?
(a) Getter and setters
(b) Method names that begin and end with two underscore characters
(c) Methods that are part of any Python built-in type

5. Which of the following is a special method for defining an operator in a class?
(a)__init__
(b)__add
(c)__add__

ANSWERS: 1. False, 2. True, 3. getters/setters, 4. (b), 5. (c)

10.3 Inheritance

The true capabilities of object-oriented programming present themselves when inheritance of classes is employed. In this section, we explore the use of inheritance in Python.

10.3.1 What Is Inheritance?

Inheritance , in object-oriented programming, is the ability of a class to inherit members of another class as part of its own definition. The inheriting class is called a **subclass** (also “derived class” or “child class”), and the class inherited from is called the **superclass** (also “base class” or “parent class”). Superclasses may themselves inherit from other classes, resulting in a hierarchy of classes as shown in Figure 10-13 (inherited class members are in gray).

Class A is the superclass of all the classes in the figure. Thus, subclasses B and E each inherit variable var1 and method method1 from Class A. In addition, Class B defines variable var2 and method method2, and Class E defines method5. Since Class C is a subclass of Class B, it inherits everything in Class A and Class B, adding var3 and method3 in its own definition. And since Class D is also a subclass of Class B, it inherits everything in Class A and Class B, also defining method4.

FIGURE 10-13 Class Hierarchy

Inheritance in object-oriented programming is the ability of a **subclass** (also “derived class” or “child class”) to inherit members of a **superclass** (also “base class” or “parent class”) as part of its own definition.

10.3.2 Subtypes

A **subtype** is something that can be substituted for and behave as its parent type (and its parent’s parent type, etc.). For example, consider the characteristic features within the Animal Kingdom in Figure 10-14.

FIGURE 10-14
Hierarchy of Characteristics of Animals

Mammals are a type of animal. Therefore, they have the characteristic of being multicellular. In addition, they have the characteristics of having hair, mammary glands, and vertebrae. Primates (which include chimpanzees) are a type of mammal (and thus a type of animal). Therefore, they have the characteristics of mammals, as well as having large brains and opposable thumbs. Proboscidea (which includes elephants) are a type of mammal (and a type of animal) and thus have the characteristics of mammals, as well as having some form of trunk and being plant eaters. Thus, each classification describes a subtype of the classifications from which it derives. For example, consider the following simple story.

Andy was very interested in animals. He had many books about them, and went to see animals whenever he had the chance.
Because chimpanzees are a type of animal, an alternate version of the story can be generated by substituting “chimpanzee” for “animal,”
Andy was very interested in chimpanzees. He had many books about them, and

went to see chimpanzees whenever he had the chance.
Now consider the following story,
Andy was very interested in chimpanzees. He had many books about them, and loved to watch the chimpanzees swing from tree to tree.
In this case, if we substitute another animal, “elephant,” for “chimpanzee,” we get the following story,
Andy was very interested in elephants. He had many books about them, and loved to watch the elephants swing from tree to tree.

Because an elephant is not a chimpanzee, this version of the story does not make sense. Note, however, that since an elephant is a type of animal, “elephant” can be substituted in the original story just as chimpanzee was and make sense. We next look at how to create subclasses in Python.

A **subtype** is something that can be substituted for and behave as its parent type (and *its* parent type, etc.).
10.3.3 Defining Subclasses in Python

We now look at how to employ the object-oriented programming feature of inheritance in Python. We give an example of an “exploded” string class as a subclass of the built-in string class, and look at whether an exploded string can be substituted as a string type.

Class Names of the Built-In Types in Python

Recall that all values in Python are objects. Thus, there exists a class definition for each of the builtin types. To determine the type (class name) of a particular value (object) in Python, the built-in function type can be used.

... type(12) ... type(12.4) type(') ,class 'int' . ,class 'float' . ,class 'str' . ... type([]) ... type(()) type({}) ,class 'list' . ,class 'tuple' . ,class 'dict' .

The resulting expression ,class *classname*. gives the associated class name for any value. (We use some empty values here, such as the empty string, but the type of any value can be determined this way.) A detailed description of a built-in class can be displayed by use of the help function,

... help(int) ... help(str) Partial display of the str built-in type is given in Figure 10-15.

```
 >>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(string[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given encoded string.
 |  encoding defaults to the current default string encoding.
 |  errors can be 'strict', 'replace' or 'ignore' and defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x
 |
 |  __eq__(...)
 |      x.__eq__(y) <==> x==y
 |
    .
    .

 |  find(...)
 |      S.find(sub[, start[, end]]) -> int
 |
 |      Return the lowest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Return -1 on failure.
    .
    .
```

FIGURE 10-15 Description of Built-in String Class

For programmer-defined classes, such as the Fraction class developed earlier, we use the name of the class (with any required arguments) for creating objects of that type,

frac1 5 Fraction(1,2)

For built-in types such as the str type, object instances are generally created using a more convenient syntax. For example, to create a new string value (object), we simply put quotes around the desired characters of the string,

name 5 'John Smith'

For creating a new list, the list elements are surrounded by square brackets,

nums 5 [10,20,30,40]

We have seen a similar means of creating tuples, dictionaries, and sets. Knowing

the class names of the built-in types, we can alternatively create object instances of each as follows,
... int(1) ... int('1') ... int(1.2) 1 1 1
... list([1, 2, 3]) ...list((1,2,3)) ... list('123') [1, 2, 3] [1, 2, 3] [1, 2, 3]

When using the class name of a built-in type to create a new object instance, arguments of various types can be used to initialize its value. For example, an integer can be created from a provided string or float value; a list can be created from a provided tuple or string. Finally, we note that the built-in class names in Python contain only lowercase letters. Programmer-defined classes, such as Fraction however, are by convention named with a beginning uppercase letter.

LET’S TRY IT
Enter the following in the Python shell and observe the results.

... type(1) ... type([]) ... help(int) ??? ??? ??? ... type(1.5) ... type([1,2,3]) ... help(float) ??? ??? ??? ... type('') ... type(()) ... help(list) ??? ??? ??? ... type('Hi') ... type((1,2,3)) ... help(tuple) ??? ???

Built-in function type can be used to determine the type (class name) of any value in Python. Built-in function help can be used to get the class description of a built-in type.
Defining an Exploded String Type

Given the built-in string class, we can easily create a new string type that is identical to the string class, and in addition provide the option of being “exploded.” By an exploded string is meant a string with spaces (blank characters) between all characters, for example, 'H e l l o'. We call this new class ExplodedStr, and give its definition in Figure 10.16.

When defining a subclass, the name of the class is followed by the name of the parent class within parentheses,
class ExplodedStr(str):

```
# Exploded String Class

class ExplodedStr(str):

    def __init__(self, value = ''):

        # call to init of str class
        str.__init__(value)

    def explode(self):

        # empty str returned unaltered
        if len(self) == 0:
            return self
        else:
            # create exploded string
            empty_str = ''
            blank_char = ' '
            temp_str = empty_str

            for k in range(0, len(self) - 1):
                temp_str = temp_str + self[k] + blank_char

            # append last char without following blank
            temp_str = temp_str + self[len(self)- 1]

            # return exploded str by joining all chars in list
            return temp_str
```

FIGURE 10-16 ExplodedStr Type as a Subclass of the Built-in str Class

Our new exploded string type can be used as shown below.

... title 5 ExplodedStr('My Favorite Movies') ... print(title.explode())
M y F a v o r i t e M o v i e s

Let’s see how this works. The ExplodedStr class does not define any instance variables of its own. Thus, its __init__ method simply calls the __init__ method of the built-in str class to pass the value that the string is to be initialized to. If an initial value is not provided, the method has a default argument assigned to the empty string,

def __init__(self, value 5 ''):
# call to init of str class
str.__init__(value)
Here str.__init__(value) is used to call the str class’s__init__ method.

Method explode returns the exploded version of the string. First, a check is made

to see if the string is the empty string. If so, then the reference self is returned, thus returning its unaltered value. Otherwise, a new string is created (temp_str), equal to the original string referenced by self, with a blank appended after every character except the last. A for loop is used for this construction,

for k in range(0, len(self) 2 1):
temp_str 5 temp_str 1 self[k] 1 blank_char

Since the range function is called with parameters 0 and len(self) – 1, all except the last character of the original string is appended by this loop (since the last character should not have a blank character appended after it). Finally, the newly constructed exploded string (temp_str) is returned.

We can place the ExplodedStr class in its own module called exploded_str and import it when needed. Testing this new string type, we see that it behaves as desired.

... reg_str 5 'Hello' defining a regular string ... ex_str 5 ExplodedStr('Hello') ...
reg_str
'Hello'
... ex_str
'Hello'
... ex_str.explode()
H e l l o
... reg_str 55 ex_str
True
...
defining an exploded string value of regular string

value of exploded string
call to explode method
comparing the strings
LET’S TRY IT For the following underlined string uStr class definition, enter and save it in a Python file, and execute. Then enter the associated instructions within the Python shell and observe the results.
class uStr(str): ...reg_str 5 'Hello' ...u_str 5 uStr('Hello') def __init__(self, u_str): str.__init__(u_str) ...reg_str

???

```
def underline(self): ...u_str
???
return str.__str__(self) 1 '\n' 1 \
format(", '2,' 1 str(len(self))) ...u_str.underline()
???
...reg_str 55 u_str
???
```

We see that exploded strings can be used as a regular string or in exploded form. Thus, it is only the added behavior of being able to be exploded that is different, not its value. As a result, the ExplodedStr subclass serves as a subtype of the built-in string class. We discuss the related issue of polymorphism next.

When defining a subclass, the name of the class is followed by the name of the parent class within parentheses.

10.3.4 Let's Apply It—A Mixed Fraction Class

The following MixedFraction class is implemented as a subclass (subtype) of the Fraction class developed earlier. The program utilizes the following programming features:

➤ Inheritance of classes

The Fraction class that we developed only represents values as common fractions, that is, with just a numerator and denominator. Thus, the value one and a half is represented as 3/2. Mixed (compound) fractions denote values with a separate whole value and (proper) fraction—3/2 is represented as 1 1/2. In certain applications, such as in the recipe conversion program developed earlier, the latter representation is preferable. An example of this is given below for a conversion of the chocolate chip cookie recipe. Example testing of the Fraction class is shown in Figure 10-17.

```
>>> frac1 = Fraction(5,4)
>>> frac1
5/4
>>> frac2 = Fraction(11,4)
>>> frac2
11/4
>>> -frac1
-5/4
>>> frac1 + frac2
4/1
>>> frac1 - frac2
-3/2
>>> frac1 * frac2
55/16
>>>
```

Use of `Fraction` Type

```
>>> frac1 = MixedFraction(5,4)
>>> frac1
1 1/4
>>> frac2 = MixedFraction(11,4)
>>> frac2
2 3/4
>>> -frac1
-1 1/4
>>> frac1 + frac2
4
>>> frac1 - frac2
-1 1/2
>>> frac1 * frac2
3 7/16
>>>
```

Use of `MixedFraction` Type

FIGURE 10-17 Interactive Testing of the MixedFraction Class

The implementation of the MixedFraction class is given in Figure 10-18. The Fraction class is imported on **line 3** using the from-import form of import. This class provides the (special) methods for performing arithmetic and relational operations on fractions. Therefore, the operations

```
# MixedFraction Class

from fraction import *

class MixedFraction(Fraction):

    def __init__(self, *args):
        if len(args) == 2:
            self.__whole_num = 0
            Fraction.__init__(self, args[0], args[1])
        elif len(args) == 3:
            self.__whole_num = args[0]
            Fraction.__init__(self, args[1], args[2])
        else:
            raise TypeError('MixedFraction takes 2 or 3 arguments ' + \
               '(' + str(len(args)) + ' given)')

    def __str__(self):
        empty_str = ''
        blank = ' '

        displayFrac = Fraction.copy(self)
        displayFrac.reduce()

        whole_num = 0
        numer = displayFrac.getNumerator()
        denom = displayFrac.getDenominator()

        if numer == 0:
            return '0'

        if denom == 1:
            return str(numer)

        if numer < 0:
            numer = abs(numer)
            sign = '-'
        else:
            sign = empty_str

        if abs(numer) > abs(denom):
            whole_num = abs(numer) // abs(denom)
            numer = abs(numer) % abs(denom)

        if whole_num == 0:
            return sign + str(numer) + '/' + str(denom)
        else:
            return sign + str(whole_num) + blank + \
                   str(numer) + '/' + str(denom)

    def __repr__(self):
        return self.__str__()
```

Special Methods

FIGURE 10-18 MixedFraction Class (*Continued*)

of addition, subtraction, multiplication, and comparison on mixed fractions can be accomplished by the inherited methods of the Fraction class.

The difference between the Fraction class and the MixedFraction class is in how fraction values are displayed. For example, a Fraction object with the value 3/2 is displayed by the

```
                                                    Getter and Setter Methods
def getWholeNum(self):
    return self.getNumerator() // self.getDenominator()


def setWholeNum(self, value):
    self.setNumerator(self.getNumerator() + \
                          value * self.getDenominator())

def set(self, whole_num, numer, denom):
    Fraction.set(self, numer + whole_num * denom, denom)


                                         Special Arithmetic Operator Methods
def __neg__(self):
    return MixedFraction(-Fraction.getNumerator(self),
                         Fraction.getDenominator(self))

def __sub__(self, rfraction):
    tempFrac = Fraction.__sub__(self, rfraction)

    return self.__createMixedFraction(tempFrac)

def __add__(self, rfraction):
    tempFrac = Fraction.__add__(self, rfraction)

    return self.__createMixedFraction(tempFrac)


def __mul__(self, rfraction):
    tempFrac = Fraction.__mul__(self, rfraction)

    return self.__createMixedFraction(tempFrac)

def __createMixedFraction(self, frac):                  Private Methods
    numer = frac.getNumerator()
    denom = frac.getDenominator()

    return MixedFraction(numer, denom)
```

FIGURE 10-18 MixedFraction Class

MixedFraction class as 1 1/2. For each object instance, however, the same values are stored— as an integer numerator and integer denominator value, and thus can be operated on the same way. How fractions are displayed is determined by the implementation of the __str__ special method. In the Fraction class, the numerator and denominator values are simply concatenated with a ‘/’ between

them. In the MixedFraction class, however, the fraction value is reconstructed in three parts, a whole number part (possibly 0), and a proper fraction part in which the numerator is less than the denominator. How the fraction is displayed depends on these three values. If, for example, the whole number part is 0, then only a proper fraction is displayed,

0 1 1/2 displayed as 1/2

If the denominator is 1, then only the numerator is displayed, 4/1 displayed as 4 Otherwise, both the whole number part and the associated proper fraction part are displayed, 5/4 displayed as 1 1/4

The implementation of special method __str__ in the MixedFraction class is on **lines 18– 49**. Variables empty_str and blank are initialized on **lines 19–20**. These are used in the construction of the mixed-fraction string. On **line 22** a copy is made of the object's value by use of the copy method of the Fraction class. This is so that the value of the fraction can be reduced to its simplest form, in preparation for the conversion of the fraction value to a whole number and proper fraction part. The reduction is performed by the reduce() method inherited from the Fraction class.

Once the (temporary) fraction object is reduced to simplest terms, it can be determined whether there is a whole number part to be displayed as part of the fraction value. This depends on whether the numerator is greater than the denominator. Thus, on **line 25**, variable whole_num is initialized to 0. The numerator and denominator values of the temporary displayFrac object are set to variables numer and denom, respectively (**lines 26–27**). What remains is to determine how the fraction value should be displayed based on these three values. On **line 29**, if variable numer is 0, then the fraction value is also 0; therefore '0' is returned (**line 30**). If the denominator in variable denom is 1, then the fraction value can be displayed as a single integer value (**lines 32–33**).

Next, special method __str__ handles the proper display of negative fractions. For example, 24/3 is stored as negative integer value 4 and positive integer value 3. When this value is displayed in mixed fraction form, the negative sign should appear as follows,

2 1 1/3

Thus, on **line 35**, a check is made to determine if the numerator value in numer

is negative. If so, then numer is set to its absolute value (**line 36**). This prevents a negative sign with the numerator from being displayed. Because a negative sign will need to appear with the whole number part of the displayed value, variable sign is set the dash character (**line 37**). If, on the other hand, the numerator is found to be nonnegative, then variable sign is set to the empty string (**line 39**).

In addition to checking the sign of the numerator, a check is made to see if the numerator is greater than the denominator (**line 41**). If so, then there needs to be a separate whole number value displayed along with the (proper) fraction value. Thus, if the numerator is found to be greater, then variable whole_num is set to the result of the integer division of the absolute value of the numerator by the absolute value of the denominator (**line 42**). Absolute values are used because the sign of the displayed result has already been determined (stored in variable sign). The numerator is then set to the remaining fractional part by use of the modulo operator (**line 43**). Thus, for 4/3, whole_ num is set to 1 (4 // 3), and numer is set to 1 (4 % 3). Finally, if the whole number part of the value is 0, then a string containing only the proper fraction is returned (**line 46**). If not 0, then a string of the form '1 1/3' is constructed and returned (**lines 48–49**).

The remaining part of the MixedFraction class provided the needed getters and setters for mixed fractions (**lines 56–65**). It also provides a set of arithmetic operators and arithmetic methods to replace those inherited from the Fraction class (**lines 69–87**). Finally, private method __createMixedFraction is defined as a private supporting method called by the arithmetic operators in the class.

Self-T est Questions

1. A class is made a subclass of another class by the use of _____.

2. Which of the following contains terms that mean the same thing?
(a) Parent class, derived class
(b) Parent class, base class
(c) Subclass, base class

3. All subclasses are a subtype in object-oriented programming. (TRUE/FALSE)
4. When defining a subclass in Python that is meant to serve as a subtype, the subtype Python keyword is used. (TRUE/FALSE)

5. Built-in function type can be used to,

(a) Determine the type of only the built-in types in Python
(b) Determine the type of only programmer-defined types (classes)
(c) Determine the type of all types

ANSWERS: 1. inheritance, 2. (b), 3. False, 4. False, 5. (c)

10.4 Polymorphism

Polymorphism is a powerful feature of object-oriented programming languages. It allows for the implementation of elegant software that is well designed and easily modified. We explore the use of polymorphism in this section.

10.4.1 What Is Polymorphism?

The word *polymorphism* derives from Greek meaning “something that takes many forms.” In objectoriented programming, **polymorphism** allows objects of different types, each with their own specific behaviors, to be treated as the same general type. For example, consider the Shape class and its subclasses given in Figure 10-19.

All Shape objects have an x, y coordinate (with corresponding getter and setter methods). In addition, Shape objects can also calculate their areas. How a shape’s area is computed, however,

FIGURE 10-19 Polymorphic Shape Class

depends on what shape it is. Thus, it is not possible to define a calcArea method in the Shape class that serves the purposes of all types of shapes. On the other hand, we want all shape types to have a calcArea method. Therefore, we add an *unimplemented* version of the calcArea method to the Shape class as shown in Figure 10-20.

```
class Shape:

    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def getXYLoc(self):
        return (self.__x, self.__y)

    def setXYLoc(self, x, y):
        self.__x = x
        self.__y = y

    def calcArea(self):
        raise NotImplementedError("Method calcArea not implemented")
```

FIGURE 10-20 Abstract Shape Class

Subclasses of the Shape class must implement the calcArea method, otherwise a NotImplementedError exception is raised. A class in which one or more methods are unimplemented (or only implemented to raise an exception) is called an **abstract class**. Figure 10-21 gives Circle, Square, and Triangle subclasses of the Shape class.

```
class Circle(Shape):

    def __init__(self, x, y, r):
        Shape.__init__(self, x, y)
        self.__radius = r

    def calcArea(self):
        return math.pi * self.__radius ** 2
```

```
class Square(Shape):

    def __init__(self, x, y, s):
        Shape.__init__(self, x, y)
        self.__side = s

    def calcArea(self):
        return self.__side ** 2
```

```
class Triangle(Shape):

    def __init__(self, x, y, s):
        Shape.__init__(self, x, y)
        self.__side = s

    def calcArea(self):
        return (self.__side ** 2) * math.sqrt(3) / 4.0
```

FIGURE 10-21 Subclasses Circle, Square, and Triangle

Each of the subclasses contains an __init__ method, in which the first two arguments provide the x,y location of the shape (within a graphics window) and the third argument indicates its size. Each first calls the __init__ method of the Shape class with arguments x,y to set its location, since the x,y values are maintained by the Shape class. Note that methods getXYLoc and setXYLoc are not defined in the subclasses, as they are inherited from the Shape class.

The size of each shape is handled differently, however. In the Circle class size is stored as the radius, and in the Square and Triangle classes it is stored as the length of each side. (The Triangle class represents only equilateral triangles, those in which each side is of the same length.) Given these classes, we can now see how polymorphism works in Python.

Suppose that there was a list of Shape objects for which the total area of all shapes combined was to be calculated,
shapes_list 5 (circle1, circle2, square1, triangle1, triangle2) Because each

implements the methods of the Shape class, they are all of a common general type, and therefore can be treated in the same way,
total_area 5 0
for shape in shapes_list:
total_area 5 total_area 1 shape.calcArea()
LET’S TRY IT

For the following class definitions, enter and save them in a single Python file, and execute. Then enter the associated instructions within the Python shell and observe the results.
class Bird(object):

def __init__(self, w):
print('__init__ of Bird Class called') self.__weight 5 w
... b1 5 BlueJay(1) ???

... b2 5 Cardinal(1.4) ???
def getWeight(self):
return str(self.__weight) 1 'ounces' ... b3 5 BlackBird(3.5) ???
def getColor(self):
raise NotImplementedError(\
'Method color not implemented') ... b1.getWeight() ???
class BlueJay(Bird):
def __init__(self, w): Bird.__init__(self, w) def getColor(self):
return 'Blue'
class Cardinal(Bird):
def __init__(self, w): Bird.__init__(self, w) def getColor(self):
return 'Red'
class BlackBird(Bird):
def __init__(self, w):

Bird.__init__(self, w) ... b2.getWeight() ???

... b3.getWeight() ???
... b1.getColor() ???
... b2.getColor() ???
... b3.getColor() ???

def getColor(self): return 'Black' ... b4 5 Bird(2.0) ... b.getColor()

In Python, it is not because Circle, Square, and Triangle are subclasses of the Shape class that allows them to be treated in a similar way. It is because the classes are *subtypes* of a common parent class. (Actually, in Python, any set of classes with a common set of methods, even if not subclasses of a common type, can be treated similarly. This kind of typing is called **duck typing—** that is, “if it looks like a duck and quacks like a duck, then it’s a duck.”)

In object-oriented programming, **polymorphism** allows objects of different types, each with their own specific behaviors, to be treated as the same general type.

10.4.2 The Use of Polymorphism

To fully appreciate the benefits of polymorphism, let’s consider the development of a program for manipulating geometric shapes. We consider a graphical environment in which classes Circle, Square, and Triangle do not have a common set of methods, and therefore cannot be treated polymorphically.

We assume that we have a graphics program in which the user selects the geometric shape that they want (stored in variable selected_shape) for which the appropriate object type is created:

```
if selected_shape 55 1: cir 5 Circle(0, 0, 1)
elif selected_shape 55 2: sqr 5 Square(0, 0, 1)
elif selected_shape 55 3: tri 5 Triangle(0, 0, 1)
```

Next, the geometric object is displayed. Since each object has its own set of methods, the appropriate method must be called. This is determined by use of another if statement:

```
if selected_shape 55 1: cir.drawCircle()
elif selected_shape 55 2: sqr.drawSquare()
elif selected_shape 55 3: tri.drawTriangle()
```

The user may then request that the area of the graphic object be displayed. Since each of the Circle, Square, and Triangle classes have a different method for calculating their area, then an if statement must again be used:

```
if selected_shape 55 1:
area 5 cir.calcCircleArea()
elif selected_shape 55 2:
```

area 5 sqr.calcSquareArea()
elif selected_shape 55 3:
area 5 tri.calcTriangleArea()

And when the graphic object is repositioned, an if statement is again needed:

if selected_shape 55 1:
cir.moveCircle(x, y)
elif selected_shape 55 2:
sqr.moveSquare(x, y)
elif selected_shape 55 3:
tri.moveTriangle(x, y)

The design of this program becomes rather tedious and inelegant. If statements abound throughout the program. Let's now look at how the same program can be written with the use of polymorphism. First, we give a more complete Shape class in Figure 10-22.

```
class Shape(object):

    def __init__(self, x, y):
       self.__x = x
       self.__y = y

    def getXYLoc(self):
       return (self.__x, self.__y)

    def setXYLoc(self, x, y):
       self.__x = x
       self.__y = y

    def draw(self):
        raise NotImplementedError("Method draw not implemented")

    def calcArea(self):
        raise NotImplementedError("Method calcArea not implemented")

    def resize(self, amt):
        raise NotImplementedError("Method resize not implemented")
```

FIGURE 10-22 Methods of Circle, Square, and Triangle Classes Following Duck Typing

As before, the unimplemented methods of the class raise a NotImplementedError

exception. In this case, each of the Circle, Square, and Triangle classes are defined as subclasses of the Shape class. Therefore, each is required to have methods for the unimplemented methods of the Shape class in order to be completely defined.

We reconsider the implementation of a graphics program for manipulating geometric shapes. We give side-by-side listing of nonpolymorphic vs. polymorphic code in Figure 10-23.

One benefit of polymorphism, as apparent from the example, is that the program code is much more straightforward and concise. Without polymorphism, there is constant use of if statements that clutters up the code. Another benefit is that it allows the programmer to think at a more abstract level ("this section of code needs to draw *some kind of geometric shape*") without having to keep in mind specific entities such as circles, squares, and triangles ("this section of code needs to draw *either a circle, square, or triangle*").

The most significant benefit of polymorphism is that programs are much easier to maintain and update. Suppose, for example, that our program had to be updated to also handle rectangles.

Non-Polymorphic Code

```
# Create Appropriate Object

if selected_shape == 1:
    cir = Circle(0, 0, 1)
elif selected_shape == 2:
    sqr = Square(0, 0, 1)
elif selected_shape == 3:
    tri = Triangle(0, 0, 1)


# draw
if selected_shape == 1:
    cir.drawCircle()
elif selected_shape == 2:
    sqr.drawSquare()
elif selected_shape == 3:
    tri.drawTriangle()

# calc area
if selected_shape == 1:
    area = cir.calcCircleArea()
elif selected_shape == 2:
    area = sqr.calcSquareArea()
elif selected_shape == 3:
    area = tri.calcTriangleArea()

# resize
if selected_shape == 1:
    cir.resizeCircle(percentage)
elif selected_shape == 2:
    sqr.resizeSquare(percentage)
elif selected_shape == 3:
    tri.resizeTriangle(percentage)

# reposition
if selected_shape == 1:
    cir.setXY(x, y)
elif selected_shape == 2:
    sqr.setPosition(x, y)
elif selected_shape == 3:
    tri.moveTo(x, y)
```

Polymorphic Code

```
# Create Appropriate Object

if selected_shape == 1:
    fig = Circle(0, 0, 1)
elif selected_shape == 2:
    fig = Square(0, 0, 1)
elif selection_shape == 3:
    fig = Triangle(0, 0, 1)


# draw
fig.draw()

# calc area
area = fig.calcArea()

# resize
fig.resize(selected_percentage)

# reposition
fig.setXYLoc(x, y)
```

For the call to method `draw()`, the method defined in the specific subclass is the actual method called.

For the call to method `calcArea()`, the method defined in the specific subclass is the actual method called.

For the call to method `resize()`, the method defined in the specific subclass is the actual method called.

For the call to method `setXYLoc()`, there is no method defined in any of the subclasses. Therefore, for each particular shape, the method of the `Shape` class is the method called.

FIGURE 10-23 Nonpolymorphic vs. Polymorphic Code

How much of the program would need to be changed? Without polymorphism, every if statement for the selection of method calls would need to be updated. With polymorphism, however, only the first if statement that creates the appropriate type object would need to be changed to include a Rectangle type. The rest of the code would remain the same, as shown in Figure 10-24.

In the end, selection needs to occur somewhere—either within the program (in the nonpolymorphic approach), or by the programming language (with the use of polymorphism).
Polymorphism allows the programmer to think at a more abstract level during program development, and supports the development of programs that are easier to maintain and update.

```
# Create Appropriate Object

if selected_shape == 1:
    fig = Circle(0, 0, 1)
elif selected_shape == 2:
    fig = Square(0, 0, 1)
elif selected_shape == 3:
    fig = Triangle(0, 0, 1)
elif selected_shape == 4:
    fig = Rectangle(0, 0, 1)
```

FIGURE 10-24 Updated Polymorphic Code for Incorporating a New Rectangle Type

Self-Test Questions

1. The term *polymorphism* in object-oriented programming refers to
(a) The ability to treat various type objects in a similar way
(b) The ability to change an object from one type to another type
(c) The ability to have multiple objects of the same type treated as one

2. The use of duck typing in Python results in
(a) More restriction on the type values that can be passed to a given method **(b)** Less restriction on the type values that can be passed to a given method

3. The *biggest* reason for the use of polymorphism in a program is
(a) There is less program code to write
(b) The program will result in a more elegant design, and thus will be easier to maintain and

update
(c) It allows the programmer to think at a more abstract level
ANSWERS: 1. (a), 2. (b), 3. (b)

10.5 Object-Oriented Design Using UML

We have mainly focused on object-oriented *programming* (OOP) . The first step

in the development of any object-oriented program, however, is the development of an appropriate **object-oriented design (OOD)** . Next we discuss a specification language for denoting an object-oriented design, referred to as the *Unified Modeling Language*.

10.5.1 What Is UML?

The **Unified Modeling Language (UML)** is a standardized design specification (modeling) language for specifying an object-oriented design. The term "Unified" comes from the fact that the language is a unification of three earlier object-oriented design modeling languages.

UML is a language-independent, *graphical* specification language. It contains numerous types of graphical diagrams for expressing various aspects of an object-oriented design. One of the most widely used graphical diagrams is called a "class diagram." A class diagram specifies the classes and their relationships of a given object-oriented design. We look at class diagrams in UML next.

UML ("Unified Modeling Language") is a standardized language-independent, graphical modeling language for specifying an object-oriented design.

10.5.2 UML Class Diagrams

In UML, **class diagrams** are used to express the *static* aspects of a design, such as the instance variables and methods of individual classes, their visibility (i.e., public or private), and various relationships between classes. Other UML diagrams, called **interaction diagrams**, are used to represent the sequence of method calls between objects during program execution (the *dynamic* aspect of a design). We omit discussion of interaction diagrams and look at the UML notation for denoting the classes of an object-oriented design.

Class diagrams in UML are used to express the static aspects of an object-oriented design. Other diagrams, called **interaction diagrams**, are used to represent the sequence of method calls between objects during program execution.

The Representation of Classes

A class is denoted in UML in three parts: a class name, a set of class attributes (instance variables), and a set of methods, as given below.

We give the UML specification for the abstract Shape class and the (concrete) Circle class in Figure 10-25. The names of unimplemented (abstract) methods are denoted in italics.

Initialization methods like __init__ in Python are named create() in UML. The types of attributes (instance variables) and the return type of methods is indicated by : ,type name., for any given type (for example, :Integer). The 1 and 2 symbols are used to specify if a given member of a class has either public (1) or private (2) access.

A class is denoted in UML in three parts: a class name, a set of class attributes (instance variables), and a set of methods.

FIGURE 10-25 UML Class Diagrams for Shape and Circle Classes

Denoting Associations between Classes

Associations are the most common relationship in UML class diagrams. An **association** between two classes indicates that the methods of one class make calls to methods of the other, as shown in Figure 10-26.

FIGURE 10-26 Association in UML

In this example, it is assumed that the classes are part of a graphical design package, in which the GraphicsWindow class creates and can manipulate a set of Shape objects. (We have left out the details of the GraphicsWindow and Shape classes in this diagram.)

The numbers above the association ends are referred to as **multiplicity**. The multiplicity of 1 at the GraphicsWindow end of the association and 0..* at the Shape end indicates that one GraphicsWindow object may be associated with any number of (zero or more) Shape objects. The “creates” label at the GraphicsWindow association end is referred to as a **role name**. Role names are used to describe an association between two classes.

Finally, the arrow denotes **navigability**. It indicates the direction of method calls made. In this example, it shows that a GraphicsWindow object makes method calls (sends messages) to Shape objects, and not the other way around.

An **association** between two classes, denoted by a connecting solid line (and a possible arrowhead) indicates that methods of one class call methods of the other.

Denoting Subclass Relationships

Subclasses are indicated in UML by use of a solid line with a closed arrow head from a subclass to its superclass, as shown in Figure 10-27.

FIGURE 10-27 UML Subclass Notation

This diagram indicates that the Circle class is a subclass of the abstract Shape class. (We again have left out the details of both the Shape and Circle classes.) Note that multiplicity is not used in subclass relationships.

Subclass relationships in UML are indicated by use of a solid line with a closed arrow head from a subclass to a superclass.
Denoting Composition vs. Aggregation

Composition indicates a “part-of” relationship between classes (Figure 10-28). The containing object is viewed as “owning” the contained object, of which the contained object is an integral part. For example, the Shape class is comprised of two integers for holding the x,y location of any Shape type. Because these two integer values are used together, we could develop an XYCoord class, in which the Shape class contains an instance. This is an example of the use of composition.

```
def class XYCoord:

    def __init__(self, x, y):
        self.__xCoord = x
        self.__yCoord = y

    def getXY(self):
       return (self.__x, self.__y)

    def setXY(self, x, y):
        self.__xCoord = x
        self.__yCoord = y
```

FIGURE 10-28 Composition of Classes in UML

When denoting composition, a filled diamond head is used at the end of the line connected to the containing class (the Shape class). With composition, it is implied that the containing class (Shape) makes calls to the member class (XYCoord), and thus composition also connotes the relationship of association.

We note that, since all values in Python are objects, *every* class with instance variables involves the use of composition, including the Shape class. However, the reason to indicate a non-built-in type as composition is that it provides a place to specify the details of the type. Instance variables of a built-in type can simply be included in the class as primitive types as was done for the x and y instances variables of the Shape class above.

Aggregation , in contrast to composition, is not a part-of relationship. It is used to denote a class that groups together, or aggregates, a set of objects that exists independently of the aggregating class. An example of aggregation is given in Figure 10-29.

```
def class ShapeCollection:

    def __init__(self):
        self.__collection = []

    def add(self, shape):
        self.__collection.append(shape)

    def remove(self, shape):
        self.__collection.remove(shape)
```

FIGURE 10-29 Aggregation of Classes in UML

Aggregation is denoted by an unfilled diamond head. Here, the ShapeCollection class contains references to an arbitrary number of Shape objects. This might be used, for example, when a graphics window allows the user to select a group of Shape objects on the screen and change an attribute of each, such as their size, all at once.

Composition is a “part of” relationship between classes denoted by a filled diamond head in UML. **Aggregation** is a “grouping” relationship, denoted by an unfilled diamond head.

An Example Class Diagram

Although UML is a specification language for modeling object-oriented software, it can be used to specify any set of entities and their relationships. Modeling everyday concepts and entities can be instructive in understanding UML. Figure 10-30 shows a UML class diagram modeling the concept of a car.

For every car, there is one engine, an integral part of a car. Thus, a composition relationship is denoted between Car and Engine, with a multiplicity of 1 on each end. As with engines, tires are an integral part of a car, so this is also indicated by composition, with a multiplicity of four tires for each passenger car.

A car is still a car with or without a driver. Therefore, a relationship of composition is not appropriate here. There is simply an association between Car

and Driver with multiplicity of 0..1 on the driver end of the association. Since a Driver is a Person, a subclass relationship is denoted between the two.

There is an association between Car and Person with a multiplicity of 0..*. Since the association denoted is not so apparent (owner? passenger?), we add the role name Passenger to the Person end of the association to be more explicit. Finally, any number of Drivers may belong to AAA (Automobile Association of America), denoted by the use of aggregation.

FIGURE 10-30 Passenger Car UML
Class Diagram

Self-Test Questions

1. Which of the following is true of UML?
(a) UML is a specification language for designing Python programs
(**b)** UML is a specification language that can be used for designing programs in various programming languages

2. In UML, class diagrams are used to express the _____ aspects of a design, and _____ are used to denote the dynamic aspects

3. In UML, an association between two classes indicates that
(a) The two classes have a common superclass

(b) Objects of each of the two class types are created at the same time
(c) Methods of one of the classes make calls to methods of the other

4. Multiplicity in UML indicates
(a) How many objects of a given class type exist
(b) How many objects of one given class there are in relation to another
(**c)** How many subclasses of a given class there may be

5. Composition in UML indicates,
(a) A “part of” relationship
(b) A grouping of objects

6. Aggregation in UML indicates,
(a) A “part of” relationship
(b) A grouping of objects

ANSWERS: 1. (b), 2. static, interaction diagrams, 3. (c), 4. (b), 5. (a), 6. (b)

COMPUTATIONAL PROBLEM SOLVING 10.6 Vehicle Rental Agency Program

In this section, we design, implement, and test a program that will serve the needs of a vehicle rental agency.

10.6.1 The Problem

The problem is to develop an object-oriented design and imple mentation of a program capable of maintaining reservations for a vehicle rental agency. The agency rents out three types of vehicles— cars, vans, and moving trucks. The program should allow users to check for available vehicles, request rental charges by vehicle type, get the cost of renting a particular type vehicle for a specified period of time, and make/cancel reservations.

10.6.2 Problem Analysis

The program needs an appropriate set of objects for the vehicle rental agency domain. An obvious class to include is a Vehicle class. It can be implemented to maintain information common to all vehicle types. Subclasses of the Vehicle class can maintain information specific to each subtype.

For example, all vehicles have a miles-per-gallon rating and a vehicle identification number (VIN). Thus, this information can be maintained in the

Vehicle class. However, there are different make and model cars (with either two or four doors, that hold a specific number of passengers); different make and model vans (able to hold a specific number of passengers); and moving trucks of various lengths, each providing a certain amount of cargo space. Therefore, the Vehicle class is made a superclass of classes Car, Van, and Truck, in which each subclass contains information (instance variables and/or methods) specific to that vehicle type.

For each type vehicle, there is a rental charge based on daily, weekly, and weekend rental rates. There is also a mileage charge and some number of free miles (on select vehicles), plus the cost of optional insurance. Because these costs are associated with particular types, but cost is not inherently *part* of a vehicle's attribute, we include a separate VehicleCost class.

Finally, we incorporate a Reservation class that maintains the information for each reservation made. This will include the customer name, address, credit card number, and the VIN of the vehicle rented.

10.6.3 Program Design

Meeting the Program Requirements

The general requirements for this program are for users to be able to check for the availability of vehicles of a certain type (cars, vans, or trucks); request rental charges by vehicle type; determine the rental cost for a particular vehicle and rental period; and make and cancel reservations. The specific requirements of this program are given in Figure 10-31.

The program must maintain a group of specific model vehicles for the following vehicle categories: cars, vans, and (moving) trucks with the following characteristics:

Cars: make/model, miles-per-gallon, num of passengers, num of doors, VIN
Vans: make/model, miles-per-gallon, number of passengers, VIN
Trucks: miles-per-gallon, length, number of rooms, VIN

The program must be able to display the specific vehicles available for rent by vehicle type.

The program must display the cost associated with a given type vehicle including daily, weekend and weekly rate, insurance cost, mileage charge, and number of free miles. It must also allow the user to determine the cost of a particular vehicle, for a given period of time, an estimated number of miles, and the cost of optional insurance.

The program must be able to allow a particular vehicle to be reserved and cancelled.

FIGURE 10-31 Program Requirements for the Vehicle Rental Agency Program

The specific rental costs for each vehicle type are given in Figure 10-32.

	Daily Rate	Weekly Rate	Weekend Rate	Free Miles Per Day	Per Mile Charge
Car	$24.99	$180.00	$45.00	100	0.15
Van	$35.00	$220.00	$55.00	0	0.20
Truck	$34.95	$425.00	$110.00	25	0.25

FIGURE 10-32 Rental Costs by Vehicle Type

The specific vehicles in stock at the rental agency are shown in Figure 10-33.

Data Description

All the data is stored as string types, converted to a numeric type when needed in a computation (such as the cost of daily insurance).

Algorithmic Approach

The algorithmic methods of the program will consist of simple search (for finding and retrieving the requested vehicle information by the user), updating of information (for marking vehicles as reserved or unreserved), and direct calculation (for calculating the total cost of a rental).

Make/Model	Mileage	Num Passengers	Num Doors	Vehicle #
CARS				
Chevrolet Camaro	30 mpg	4	2	WG8JM5492DY
Chevrolet Camaro	30 mpg	4	2	KH4GM4564GD
Ford Fusion	34 mpg	5	4	AB4FG5689GM
Ford Fusion Hybrid	35 mpg	5	4	GH2KL4278TK
Ford Fusion Hybrid	32 mpg	5	4	KU4EG3245RW
Chevrolet Impala	36 mpg	6	4	QD4PK7394JI
Chevrolet Impala	30 mpg	6	4	RK3BM4256YH

Make/Model	Mileage	Num Passengers	Vehicle #
Vans			
Chrysler Town&Country	25 mpg	7	DK3KG8312UE
Chrysler Town&Country	25 mpg	7	VM9RE2645TD
Chrysler Town&Country	25 mpg	7	WK8BF4287DX
Dodge Caravan	25 mpg	7	QK3FL4278ME
Dodge Caravan	25 mpg	7	KY8EW2053XT
Ford Expedition	20 mpg	8	JK2RT8364HY
Ford Expedition	20 mpg	8	KH4ME4216XW

Make/Model	Mileage	Cargo Space	Vehicle #
Trucks			
Ten-Foot	12 mpg	1 bedroom	EJ5KU2435BC
Ten-Foot	12 mpg	1 bedroom	KF8JP7293EK
Seventeen-Foot	10 mpg	2 bedrooms	KG4DM5472RK
Seventeen-Foot	10 mpg	2 bedrooms	PR8JH4893WQ
Twenty-Four-Foot	8 mpg	4 bedrooms	EP2WR3182QB
Twenty-Four-Foot	8 mpg	4 bedrooms	TY3GH4290EK
Twenty-Four-Foot	8 mpg	4 bedrooms	KU9FL4235RH

FIGURE 10-33 Specific Vehicles of the Vehicle Rental Agency

Overall Program Steps

The overall steps in this program design are given in Figure 10-34.

UML Class Diagram

We give a UML class diagram for the program in Figure 10-35. In addition to the “domain objects” that we have decided on in our analysis, we add a text-based user interface, provided by the RentalAgencyUI class.

Three classes store the information in the system— Vehicle (and its subclasses), VehicleCost, and Reservation. For each of these classes there is a corresponding aggregator class— Vehicles, VehicleCosts, and Reservations—that maintains a collection of the corresponding object type. Each aggregator class has methods

for maintaining its collection of objects (for example, addVehicle in the Vehicles class and addVehicleCost in the VehicleCosts class).

FIGURE 10-34
Overall Design of the Vehicle Rental Agency Program

The Vehicle class has three subclasses—Car, Van, and Truck. It is responsible for maintaining a vehicle's type, its VIN, and its reservation status. Method getDescription is provided in the Vehicle class to return the information common to all vehicles: miles per gallon, and a VIN. Each subclass builds on this inherited method to include the specific information for that vehicle type. The Car class stores the maximum number of passengers and number of doors, the Van class stores the maximum number of passengers, and the Truck class stores its length and the number of rooms of storage it can hold.

The VehicleCost class does not have any subclasses. Its create (__init__) method

is passed six arguments: the daily/weekly/weekend rates, the number of free miles, the per mile charge, and the daily insurance rate to initialize the object with. The getVehicleCost method of the VehicleCosts aggregating class returns the cost of a specified vehicle type as a single string for display. The Reservation and corresponding Reservations aggregator class are designed in a similar manner.

Finally , a SystemInterface class provides all the methods that any user interface would need for interacting with the system. Such a set of methods is referred to as an API— *Application Programming Interface*. Thus, the SystemInterface object is created first. It then reads all the vehicle rental agency data from text files VehiclesStock.txt and RentalCost.txt and populates the corresponding objects. Then, an instance of the RentalAgencyUI is created and initialized with a reference to the system interface. The only public method of the

FIGURE 10-35 Class Diagram for Vehicle Rental Agency Program

RentalAgencyUI class, start, is called to start the console interaction. The main menu for the program is given in Figure 10-36.

Example use of the system is shown in Figure 10-37. (For the sake of space, the main menu is not repeatedly shown before each selection as in the actual program execution.)

```
<<< MAIN MENU >>>
1 - Display vehicle types
2 - Check rental costs
3 - Check available vehicles
4 - Get cost of specific rental
5 - Make a reservation
6 - Cancel a reservation
7 - Quit

Enter:
```

FIGURE 10-36 Text-Based (Console) Interface for the Vehicle Rental Agency Program

Display Vehicle Types

```
*****************************************************
  * Welcome to the Friendly Vehicle Rental Agency *
*****************************************************

<<< MAIN MENU >>>
1 - Display vehicle types
2 - Check rental costs
3 - Check available vehicles
4 - Get cost of specific rental
5 - Make a reservation
6 - Cancel a reservation
7 - Quit

Enter: 1
---------------- Types of Vehicles Available for Rent ----------------

1 - Car
2 - Van
3 - Truck
-----------------------------------------------------------------------
```

Display Rental Fees for a Given Type Vehicle

```
Enter: 2
Enter type of vehicle
1 - Car
2 - Van
3 - Truck

Enter: 1
---------------------- Rental Charges for Cars -----------------------

                        Free   Per Mile  Daily
Daily  Weekly  Weekend  Miles  Charge    Insurance
24.99  180.00  45.00    100    .15       14.99
-----------------------------------------------------------------------
```

Check for Available Vehicles

```
Enter: 3
Enter type of vehicle
1 - Car
2 - Van
3 - Truck

Enter: 1
-------------------------- Available Cars ---------------------------
Chevrolet Camaro     passengers: 4   doors: 2    mpg: 30   vin: WG8JM5492DY
Chevrolet Camaro     passengers: 4   doors: 2    mpg: 30   vin: KH4GM4564GD
Ford Fusion          passengers: 5   doors: 4    mpg: 34   vin: AB4FG5689GM
Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: GH2KL4278TK
Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: KU4EG3245RW
Chevrolet Impala     passengers: 6   doors: 4    mpg: 30   vin: QD4PK7394J1
```

Get the Cost of a Particular Rental

```
Enter: 4
Enter type of vehicle
1 - Car
```

FIGURE 10-37 Example Program Execution (*Continued*)

```
2 - Van
3 - Truck
Enter: 1
Enter the rental period:
1 - Daily   2 - Weekly  3 - Weekend

Enter: 1
How many days do you need the vehicle? 3
Would you like the insurance? (y/n): n
Number of miles expect to drive?: 240

--------------------- ESTIMATED Car RENTAL COST ----------------------
* You have opted out of the daily insurance *

Daily rental for 3 days would be $ 74.97

Your cost with an estimated mileage of 240 would be 95.97
which includes 100 free miles and a charge of 0.15 per mile
----------------------------------------------------------------------
```

Reserve a Particular Vehicle

```
Enter: 5

Enter type of vehicle
1 - Car
2 - Van
3 - Truck

Enter: 1
--------------------------- Available Cars ---------------------------
1-Chevrolet Camero     passengers: 4   doors: 2    mpg: 30   vin: WG8JM5492DY
2-Chevrolet Camero     passengers: 4   doors: 2    mpg: 30   vin: KH4GM4564GD
3-Ford Fusion          passengers: 5   doors: 4    mpg: 34   vin: AB4FG5689GM
4-Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: GH2KL4278TK
5-Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: KU4EG3245RW
6-Chevrolet Impala     passengers: 6   doors: 4    mpg: 30   vin: QD4PK7394J1
7-Chevrolet Impala     passengers: 6   doors: 4    mpg: 30   vin: RK3BM4256YH

Enter number of vehicle to reserve: 4
Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: GH2KL4278TK

Enter first and last name: John Smith
Enter address: 123 Main Street, Sometown 12345
Enter credit card number: 204881605
* Reservation Made *
```

Cancel a Reservation

```
Enter: 6
Please enter your credit card number: 204881605

RESERVATION INFORMATION
Name: John Smith 123 Main Street, Sometown 21252
Vehicle: Chevrolet Camaro     passengers: 4   doors: 2    mpg: 30   vin: KH4GM4564GD

Confirm Cancellation (y/n): y
* Reservation Cancelled *


Enter: 7
Thank you for using the Friendly Rental Agency
```

FIGURE 10-37 Example Program Execution

10.6.4 Program Implementation and Testing

From the UML specification, we implement and test the program. We start with the implementation of the Vehicle class, given in Figure 10-38. The classes will be placed in their own file, and imported into the program.

Development and Testing of the Vehicle, Car, Van, and Truck Classes

In the Vehicle class, the __init__ method (**lines 10–15**) defines the attributes common to each subclass—mpg, vin, and reserved. There is only one setter method, setReserved, since the other values are defined when the object is created. Other methods include getters getType (**line 17**), getVin (**line 22**), getDescription (**line 27**), and Boolean isReserved (**line 35**).

```python
"""This module provides a Vehicle class. """

class Vehicle:
    """The Vehicle class holds the mpg, vin and reserved flag of a vehicle.

       Contains attributes common to all vehicles: mpg, vin, and reserved
       (Boolean). Provides polymorphic behavior for method getDescription.
    """

    def __init__(self, mpg, vin):
       """Initializes a Vehicle object with mpg and vin."""

        self.__mpg = mpg
        self.__vin = vin
        self.__reserved = False

    def getType(self):
        """Returns the type of vehicle (car, van, truck.)"""

        return type(self).__name__

    def getVin(self):
        """Returns the vin of vehicle."""

        return self.__vin

    def getDescription(self):
        """Returns general description of car, not specific to type."""

        descript = 'mpg:' + format(self.__mpg, '>3') + '   ' + \
                   'vin:' + format(self.__vin, '>12')

        return descript

    def isReserved(self):
        """Returns True if vehicle is reserved, otherwise returns False."""

        return self.__reserved

    def setReserved(self, reserved):
        """Sets reserved flag of vehicle to provided Boolean value."""

        self.__reserved = reserved
```

FIGURE 10-38 Vehicle Class

The getType methods returns the specific type of a Vehicle object (Car, Van, or Truck) by use of type(self).__name__ which returns its own type.

We can execute the file and interactively and unit test the class in the Python shell.

... v 5 Vehicle('32', 'ABC123') ... v.getType()
'Vehicle'
... v.getVin()
'ABC123'

— create instance
— get object type
— get vehicle identification number We test the setReserved and isReserved methods similarily. An implementation of the Car subclass is given in Figure 10-39.

```
1   """This module provides a Car class, a subtype of the Vehicle class."""
2
3   from vehicle import Vehicle
4
5   class Car(Vehicle):
6       """This class is a subtype of the Vehicle class.
7
8          Contains additional attributes of make and model, num of passengers, and
9          num of doors. Supports polymorphic behavior of method getDescription.
10      """
11
12      def __init__(self, make_model, mpg, num_passengers, num_doors, vin):
13          """Initialized with provided parameter values."""
14
15          super().__init__(mpg, vin)
16
17          self.__make_model = make_model
18          self.__num_passengers = num_passengers
19          self.__num_doors = num_doors
20
21      def getDescription(self):
22          """Returns description of car as a formatted string."""
23
24          spacing = '    '
25          descript = format(self.__make_model, '<18') + spacing + \
26                     'passengers: ' + self.__num_passengers + spacing + \
27                     'doors: ' + format(self.__num_doors, '<2') + spacing + \
28                     Vehicle.getDescription(self)
29
30          return descript
```

FIGURE 10-39 Car Subclass of the Vehicle Class

The Car class is defined as a subclass of the Vehicle class, imported on **line 3**. The methods inherited from the Vehicle class have already been tested. We therefore test method getDescription (**lines 21–30**) by executing the Car class file and performing the following.

... v 5 Car('Ford Fusion', '34', '5', '4', 'AB4FG5689GM') ... v.getDescription() 'Ford Fusion passengers: 5 doors: 4 mpg: 34 vin: AB4FG5689GM'

These results are correct. Figure 10-40 and Figure 10-41 show similar implementations of the Van and Truck classes.

The Van and Truck classes differ from the Car class in the particular instance

variables they contain, and the information returned by method getDescription. These differences support the polymorphic behavior of Vehicle types.

Development and Testing of the Vehicles Class

Whereas the Vehicle class represents the information of a single vehicle, the Vehicles class maintains a complete collection of Vehicle types. Figure 10-42 illustrates an implementation of the Vehicles class, maintaining the cost, availability, and reservation status of all vehicles of the rental agency.

```
"""This module provides a Van class, a subtype of the Vehicle class."""

from vehicle import Vehicle

class Van(Vehicle):

    """This class is a subtype of the Vehicle class.

       Contains additional attributes of make and mode, and num of passengers.
       Supports polymorphic behavior of method getDescription.
    """
    def __init__(self, make_model, mpg, num_passengers, vin):
        """ Initializes with make-model, mpg, num_passengers, vin."""

        super().__init__(mpg, vin)

        self.__make_model = make_model
        self.__num_passengers = num_passengers

    def getDescription(self):
        """Returns complete description of van"""

        spacing = '  '
        descript = format(self.__make_model, '<22') + spacing + \
                   'passengers:' + format(self.__num_passengers, '>2') + \
                   spacing + Vehicle.getDescription(self)

        return descript
```

FIGURE 10-40 Van Subclass of the Vehicle Class

```python
"""This module provides a Truck class. a subtype of the Vehicle class."""

from vehicle import Vehicle

class Truck(Vehicle):

    """This class is a subtype of the Vehicle class.

       Contains additional attributes length and num of rooms storage capacity.
       Supports polymorphic behavior of method getDescription.
    """
    def __init__(self, mpg, length, num_rooms, vin):
        """ Initializes with mpg, length, num_rooms, vin."""

        super().__init__(mpg, vin)

        self.__length = length
        self.__num_rooms = num_rooms

    def getDescription(self):
        """Returns complete description of truck."""

        spacing = '   '
        descript = 'length(feet):' + format(self.__length, '>3') + spacing + \
                   'rooms:' + format(self.__num_rooms, '>2') + spacing + \
                   Vehicle.getDescription(self)

        return descript
```

FIGURE 10-41 Truck Subclass of the Vehicle Class

```
1   """Vehicles class module. Raises InvalidFileFormatError and InvalidVinError."""
2
3   from vehicle import Vehicle
4   from car import Car
5   from van import Van
6   from truck import Truck
7
8   class InvalidVinError(Exception):
9       """Exception indicating that a provided vin was not found."""
10      pass
11
12
13  class Vehicles:
14      """This class maintains a collection of Vehicle objects."""
15
16      def __init__(self):
17          """Initializes empty list of vehicles."""
18
19          self.__vehicles = []
20
21      def getVehicle(self, vin):
22          """Returns Vehicle for provided vin. Raises InvalidVinError."""
23
24          for vehicle in self.__vehicles:
25              if vehicle.getVin() == vin:
26                  return vehicle
27
28          raise InvalidVinError
29
30      def addVehicle(self, vehicle):
31          """Adds new vehicle to list of vehicles."""
32
33          self.__vehicles.append(vehicle)
34
35      def numAvailVehicles(self, vehicle_type):
36          """Returns number of available vehicles of vehicle_type."""
37
38          return len(self.getAvailVehicles(vehicle_type))
39
40      def getAvailVehicles(self, vehicle_type):
41          """Returns a list of unreserved Vehicles objects of vehicle_type."""
42
43          return [veh for veh in self.__vehicles \
44                  if veh.getType() == vehicle_type and not veh.isReserved()]
45
46      def unreserveVehicle(self, vin):
47          """Sets reservation status of vehicle with vin to unreserved."""
48
49          k = 0
50          found = False
51
52          while not found:
53              if self.__vehicles[k].getVin() == vin:
54                  self.__vehicles[k].setReserved(False)
55                  found = True
```

FIGURE 10-42 Vehicles Class

The Vehicles class imports classes Vehicle, Car, Van, and Truck (**lines 3–6**). An

InvalidVinError exception class is defined (**lines 8–10**), raised when method getVehicle is called for a nonexistent VIN. Method addVehicle (**lines 30–33**) adds a new vehicle to the collection. This is called when the Vehicles object is initially populated at the start of the program (in the SystemInterface class). Method getAvailVehicles (**lines 40–44**) returns a list of vehicles that are not currently reserved.

We can easily test these classes from the Python shell as given below.

First, the needed classes are imported: ... from vehicle import Vehicle ... from car import Car
... from van import Van
... from truck import Truck ... from vehicles import Vehicles

Then, an instance of each type Vehicle is created: ... veh_1 5 Car('Ford Fusion', '34', '5', '4', 'FG1000') ... veh_2 5 Van('Dodge Caravan', '25', '7', 'TF1000') ... veh_3 5 Truck('12', '10', '1', 'HG1000')

A Vehicles instance is created, and each Vehicle instance is added to the collection: ... v 5 Vehicles()
... v.addVehicle(veh_1)
... v.addVehicle(veh_2)
... v.addVehicle(veh_3)

Finally, the description of each vehicle is obtained: ... v.getAvailVehicles('Car')[0].getDescription()
'Ford Fusion passengers: 5 doors: 4 mpg: 34 vin: FG1000'

... v.getAvailVehicles('Van')[0].getDescription()
'Dodge Caravan passengers: 7 mpg: 25 vin: TF1000' ... v.getAvailVehicles('Truck')[0].getDescription()
'length(feet): 10 rooms: 1 mpg: 12 vin: HG1000' ...

Method getAvailVehicles returns a list of available cars for the provided vehicle type. Since this returns a list of vehicles, the first vehicle in the list is selected (at index [0]), and the getDescription method is called to display the full description of the vehicle. We see that we get the correct results.

Implementation of the **VehicleCost** and **VehicleCosts** Classes The VehicleCosts class is responsible for maintaining and providing the costs for each vehicle

type. First, the VehicleCost class is given in Figure 10-43.

```
1   """This module provides a VehicleCost class."""
2
3   class VehicleCost:
4       """This class provides the methods for maintaining rental costs."""
5
6       def __init__(self, daily_rate, weekly_rate, weekend_rate,
7                    free_miles, per_mile_chrg, insur_rate):
8           """Initialzes rental rates, num free miles /mileage chrg/insur rate."""
9
10          self.__daily_rate = daily_rate
11          self.__weekly_rate = weekly_rate
12          self.__weekend_rate = weekend_rate
13          self.__free_miles = free_miles
14          self.__per_mile_chrg = per_mile_chrg
15          self.__insur_rate = insur_rate
16
17      def getDailyRate(self):
18          """Returns daily rental rate of vehicle."""
19
20          return float(self.__daily_rate)
21
22      def getWeeklyRate(self):
23          """Returns weekly rental rate of vehicle."""
24
25          return float(self.__weekly_rate)
26
27      def getWeekendRate(self):
28          """Returns weekend rental rate of vehicle."""
29
30          return float(self.__weekend_rate)
31
32      def getFreeMiles(self):
33          """Returns number of free miles for vehicle rental."""
34
35          return int(self.__free_miles)
36
37      def getPerMileCharge(self):
38          """Returns per mile charge for vehicle rental."""
39
40          return float(self.__per_mile_chrg)
41
42      def getInsuranceRate(self):
43          """Returns daily insurance rate for vehicle."""
44
45          return float(self.__insur_rate)
46
47      def getCosts(self):
48          """Returns a list containing all costs for vehicle."""
49
50          return [self.__daily_rate, self.__weekly_rate,
51                  self.__weekend_rate, self.__free_miles,
52                  self.__per_mile_chrg, self.__insur_rate]
```

FIGURE 10-43 VehicleCost Class

The VehicleCost class, like the Vehicle class, stores information accessed by the provided getter methods. The information to be stored is passed to the __init__ method (**lines 6–15**). The getCosts method (**lines 47–52**) returns all the

individual cost components as a list. This method is called when the vehicle costs need to be displayed, or the cost of a particular vehicle rental needs to be determined.
We can easily test the VehicleCost class from the Python shell as given below.

First, the VehicleCost class is imported: ... from vehicleCost import VehicleCost
Then, an instance of each type VehicleCost is created: ... vc 5 VehicleCost('24.99', '180.00', '45.00', '100', '.15', '14.99')
Finally, each getter method is tested:

```
... vc.getDailyRate()
24.99
... vc.getWeeklyRate()
180.0
... vc.getWeekendRate()
45.0
... vc.getFreeMiles()
100
... vc.getPerMileCharge()
0.15
... vc.getInsuranceRate()
14.99
... vc.getCosts()
['24.99', '180.00', '45.00', '100', '.15', '14.99'] ...
```

We see that we get the correct results.
In Figure 10-44, we give the VehicleCosts class that maintains the collection of VehicleCost objects.
The vehicleCosts class, as with the Vehicles class, maintains a collection of VehicleCost objects. The VehicleCost class is imported on **line 3**. On **lines 6–8** three symbolic
constants are defined, DAILY_RENTAL, WEEKLY_RENTAL, and WEEKEND_RENTAL. The use of
these constants (equal to integer values 1, 2, and 3, respectively) make the program more readable
than if the corresponding integer values were used.
The VehicleCosts class is defined on **lines 10–89**. The vehicle costs for the three types of vehicles (cars, vans, and trucks) are stored in a dictionary using the vehicle types as

key values. The __init__ method (**lines 13–16**), therefore, initializes instance variable
vehicle_costs to an empty dictionary. The vehicle costs are individually added to the collection of costs by method addVehicleCost (**lines 23–26**), called from the SystemInterface class when populating the rental costs from file. Method getVehicleCost
(**lines 18–21**) returns the costs as a list of individual costs. Finally, method calcRentalCost
(**lines 28–89**) calculates the cost of a particular rental. We test the VehicleCosts class from
the Python shell.

```python
"""This module provides a VehicleCosts class"""

from vehicleCost import VehicleCost

# symbol constants
DAILY_RENTAL = 1
WEEKLY_RENTAL = 2
WEEKEND_RENTAL = 3

class VehicleCosts:
    """This class provides the methods for maintaining rental costs."""

    def __init__(self):
        """Initializes vehicle costs to empty."""

        self.__vehicle_costs = dict()

    def getVehicleCost(self, vehicle_type):
        """Returns VehicleCost object for the specified vehicle type."""

        return self.__vehicle_costs[vehicle_type]

    def addVehicleCost(self, veh_type, veh_cost):
        """Adds a vehicle cost object to dictionary with keyword veh_type."""

        self.__vehicle_costs[veh_type] = veh_cost

    def calcRentalCost(self, vehicle_type, rental_period,
                       want_insurance, miles_driving):
        """Returns estimate of rental cost for provided parameter values.

           Returns dictionary with key values: {'base_charges', 'insur_rate',
           'num_free_miles', 'per_mile_charge', 'estimated_mileage_charges'}
        """

        # get vehicle cost
        vehicle_cost = self.getVehicleCost(vehicle_type)

        # calc rental charges
        rental_time = rental_period[1]

        if rental_period[0] == DAILY_RENTAL:
            rental_rate = vehicle_cost.getDailyRate()
            rental_period_value = DAILY_RENTAL
            rental_period_str = 'daily rental'
            rental_days = rental_time
        elif rental_period[0] == WEEKLY_RENTAL:
            rental_rate = vehicle_cost.getWeeklyRate()
            rental_period_value = WEEKLY_RENTAL
            rental_period_str = 'weekly rental'
            rental_days = rental_time * 7
        elif rental_period[0] == WEEKEND_RENTAL:
            rental_rate = vehicle_cost.getWeekendRate()
            rental_period_value = WEEKEND_RENTAL
            rental_period_str = 'weekend rental'
            rental_days = 2

```

FIGURE 10-44 VehicleCosts Class (*Continued*)

```
        elif rental_period[0] == WEEKEND_RENTAL:

            rental_rate = vehicle_cost.getWeekendRate()
            rental_period_value = WEEKEND_RENTAL
            rental_period_str = 'weekend rental'
            rental_days = 2

        # get free miles, per mile charge and insurance rate
        num_free_miles = vehicle_cost.getFreeMiles()
        per_mile_charge = vehicle_cost.getPerMileCharge()
        insurance_rate = vehicle_cost.getInsuranceRate()

        # calc rental charge for selected rental period
        if not want_insurance:
            insurance_rate = 0

        # calc base rental charges
        base_rental_charges = rental_days * rental_rate + \
                              rental_days * insurance_rate

        miles_charged = miles_driving - num_free_miles

        if miles_charged < 0:
            miles_charged = 0

        estimated_mileage_charges = miles_charged * per_mile_charge

        return {'base_charges' : base_rental_charges,
                'insur_rate' : insurance_rate,
                'num_free_miles' : num_free_miles,
                'per_mile_charge' : per_mile_charge,
                'estimated_mileage_charges' : estimated_mileage_charges}
```

FIGURE 10-44 VehicleCosts Class

First, the VehicleCosts class is imported:
... from vehicleCosts import VehicleCosts
Then, instances of type VehicleCost are created:

... vc_1 5 VehicleCost('24.99', '180.00', '45.00', '100', '.15', '14.99') ... vc_2 5 VehicleCost('35.00', '220.00', '55.00', '0', '.20', '14.99') ... vc_3 5 VehicleCost('55.00', '425.00', '110.00', '25', '.25', '24.99')

Then a VehicleCosts instance is created, and each VehicleCost instance is added to the collection:

... vc.addVehicleCost('Car', vc_1)
... vc.addVehicleCost('Van', vc_2)
... vc.addVehicleCost('Truck', vc_3)

Then, each getter method is tested:

```
... vc.getVehicleCost('Car').getCosts()
['24.99', '180.00', '45.00', '100', '.15', '14.99']
... vc.getVehicleCost('Van').getCosts()
['35.00', '220.00', '55.00', '0', '.20', '14.99']
... vc.getVehicleCost(‘Truck').getCosts()
['55.00', '425.00', '110.00', '25', '.25', '24.99']
```

Finally, method calcRentalCost is tested:

```
... vc.calcRentalCost('Car', (1, 3), False, 150)
{'insur_rate': 0, 'base_charges': 74.97, 'num_free_miles': 100, 'per_mile_charge':
0.15, 'estimated_mileage_charges': 7.5}
```

In the call to calcRentalCost, recall that the first parameter is the vehicle type, the second is the rental period (1 for DAILY_RENTAL) and the number of days (3), the third indicates if the insurance is desired, and the fourth is the expected number of miles to be driven.

We see that we get the correct results. We next give the implementation of the Reservation and Reservations classes.

Implementation of the Reservation and Reservations Classes

The remaining aggregating class in the program is the Reservations class. We first give the Reservation class in Figure 10-45.

The Reservation class maintains the information for a given reservation. The __init__ method (**lines 6–14**) is provided a name, address, credit card number, and VIN when a new Reservation object is created, with a getter methods provided for each of these four values. We omit the testing of this class and next give the implementation of the corresponding Reservations class in Figure 10-46.

```python
"""This module contains a Reservation class for storing details of a rental"""

class Reservation:
    """This class provides the methods for maintaining reservations info."""

    def __init__(self, name, address, credit_card, vin):
        """Initialzes a Reservation object with name, address, credit_card
           and vin.
        """

        self.__name = name
        self.__address = address
        self.__credit_card = credit_card
        self.__vin = vin

    def getName(self):
        """Returns first and last name for reservation."""

        return self.__name

    def getAddress(self):
        """Returns address for reservation."""

        return self.__address

    def getCreditCard(self):
        """Returns credit card on reservation."""

        return self.__credit_card

    def getVin(self):
        """Returns vehicle identification number on reservation."""

        return self.  vin
```

FIGURE 10-45 Reservation Class

```
1  """This module provides a Reservations class."""
2
3  from reservation import Reservation
4
5  class Reservations:
6      """This class provides the methods for maintaiing rental reservations."""
7
8      def __init__(self):
9          """Initializes empty collection of reservations."""
10
11         self.__reservations = dict()
12
13     def isReserved(self, vin):
14         """Returns True if reservation for vin, else returns False."""
15
16         return vin in self.__reservations
17
18     def getVinForReserv(self, credit_card):
19         """Returns vin of vehicles reserved with credit_card."""
20
21         return self.__reservations[credit_card].getVin()
22
23     def addReservation(self, resv):
24         """Adds new reservation."""
25
26         self.__reservations[resv.getCreditCard()] = resv
27
28     def findReservation(self, credit_card):
29         """Returns True if reservation for credit_card, else returns False."""
30
31         return credit_card in self.__reservations
32
33     def cancelReservation(self, credit_card):
34         """Deletes reservation matching provided credit card number."""
35
36         del(self.__reservations[credit_card])
```

FIGURE 10-46 Reservations Class

The Reservation class is imported on **line 3**. The __init__ method initializes an empty dictionary for storing the reservations, with credit card numbers serving as the key values. The remaining methods of the class include method isReserved (**lines 13–16**), getVinForReserv (**lines 18–21**), addReservation (**lines 23–26**), findReservation (**lines 28–31**), and CancelReservation (**lines 33–36**). We omit the testing of the Reservations class. We finally look at the implementation of the SystemInterface and RentalAgencyUI classes.

Implementation of the **SystemInterface** and **RentalAgencyUI** Classes An implementation of the SystemInterface class is given in Figure 10-47.

The docstring for the module specifies the formatting of the files storing the vehicle and vehicle rental cost information. There is one exception raised by the

module to inform the user interface when a file error has occurred. This could be due to either a file not being found, or an improperly formatted file.

The import statements on **lines 28–30** import most of the classes in the system. Symbolic constants VEHICLE_TYPES, VEHICLE_FILENAME, and VEHICLE_COSTS_FILENAME are defined (**lines 33–35**). Defining these constants makes the program more readable and modifiable. Exception class InvalidFileFormatError is defined (**lines 39–49**) and used within the SystemInterface class. Having this exception type allows typical I/O errors (such as a file not found error) to be reported as errors specific to the file formatting requirements of the program. The exception class defines special method __str__ to enable information to be displayed specific to the error (that is, which file header was expected and not found, in which file).

The __init__ method (**lines 55–77**) first creates instances of the three aggregator classes of the system—Vehicles, VehicleCosts, and Reservations. Each is initially empty when created. The rest of the __init__ method consists of a try block that attempts to open and read the files defined by VEHICLES_FILENAME and VEHICLE_COSTS_FILENAME. If opened successfully,

```
"""This module provides a SystemInterface class for the Vehicle Rental Program.

   Vehicles File Format
   --------------------
   The format for the vehicles file contains comma-separated values with the
   indicated header lines for cars, vans, and trucks.

     #CARS#
     make-model, mpg, num-passengers, vin
     .
     #VANS#
     make-model, mpg, num passengers, vin
     .
     #TRUCKS#
     mpg, length, num rooms, vin
     .


   Vehicle Cost File Format
   ------------------------
   The format for the rental costs file includes a header line, followed
   by three lines of comma-separated values for cars, vans and trucks.

     daily, weekly, weekend, free miles, mileage charge, insurance

   The following exceptions are raised: IOError.
"""
from vehicles import Vehicles, Car, Van, Truck
from vehicleCosts import VehicleCost, VehicleCosts
from reservations import Reservations

# symbolic constants
VEHICLE_TYPES = ('Car', 'Van', 'Truck')
VEHICLES_FILENAME = 'VehiclesStock.txt'
VEHICLE_COSTS_FILENAME = 'RentalCost.txt'

# exception class

class InvalidFileFormatError(Exception):
    """Exception indicating invalid file in file_name."""

    def __init__(self, header, file_name):
        self.__header = header
        self.__file_name = file_name

    def __str__(self):

        return 'FILE FORMAT ERROR: File header ' + self.__header + \
               ' expected in file ' + self.__file_name

```

FIGURE 10-47 SystemInterface Class (*Continued*)

```
51  class SystemInterface:
52      """This class provides the system interface of the vehicle rental
53         system.
54      """
55      def __init__(self):
56          """Populates vehicles and rental costs from file. Raises IOError."""
57
58          self.__vehicles = Vehicles()
59          self.__vehicle_costs = VehicleCosts()
60          self.__reservations = Reservations()
61          self.__vehicle_info_file = None
62
63          try:
64              self.__vehicle_info_file = open(VEHICLES_FILENAME, 'r')
65              self.__rental_cost_file = open(VEHICLE_COSTS_FILENAME, 'r')
66
67              self.__populateVehicles(self.__vehicle_info_file)
68              self.__populateCosts(self.__rental_cost_file)
69          except InvalidFileFormatError as e:
70              print(e)
71              raise IOError
72          except IOError:
73              if self.__vehicle_info_file == None:
74                  print('FILE NOT FOUND:', VEHICLES_FILENAME)
75              else:
76                  print('FILE NOT FOUND:', VEHICLE_COSTS_FILENAME)
77              raise IOError
78
79      def numAvailVehicles(self, vehicle_type):
80          """Returns the number of available vehiles. Returns 0 if no
81             no vehicles available.
82          """
83
84          return self.__vehicles.numAvailVehicles(vehicle_type)
85
86      def getVehicle(self, vin):
87          """Returns Vehicle type for given vin."""
88
89          return self.__vehicles.getVehicle(vin)
90
91      def getVehicleTypes(self):
92          """Returns all vehicle types as a tuple of strings."""
93
94          return VEHICLE_TYPES
95
96      def getVehicleCosts(self, vehicle_type):
97          """Returns vehicle costs for provided vehicle type as a list.
98
99             List of form [daily rate, weekly rate, weekend rate,
100                           num free miles, per mile charge, insur rate]
101         """
102
103         return self.__vehicle_costs.getVehicleCost(vehicle_type).getCosts()
104
```

FIGURE 10-47 SystemInterface Class (*Continued*)

then each is read to populate the corresponding object. The exception handler catches two types of exceptions, InvalidFileFormatError and IOERROR, as mentioned.

A set of getter methods is provided (**lines 86–103**) for retrieving vehicle, vehicle

type, and vehicle cost information. Method getAvailVehicles returns the list of vehicles that are not currently reserved. (Note that the list returned is constructed by the use of a list comprehension.) The list of unreserved vehicles is of a specified type. Four additional methods are provided (**lines 113–132**) for maintaining reservation information (isReserved, findReservation,

```
107     def getAvailVehicles(self, vehicle_type):
108         """Returns a list of descriptions of unreserved vehicles."""
109 
110         avail_vehicles = self.__vehicles.getAvailVehicles(vehicle_type)
111         return [veh for veh in avail_vehicles]
112 
113     def isReserved(self, vin):
114 
115         return self.__reservations.isReserved(vin)
116 
117     def findReservation(self, credit_card):
118 
119         return self.__reservations.findReservation(credit_card)
120 
121     def addReservation(self, resv):
122         """Creates reservation and marks vehicles as reserved."""
123 
124         self.__reservations.addReservation(resv)
125 
126     def cancelReservation(self, credit_card):
127         """Cancels reservation made with provided credit card."""
128 
129         vin = self.__reservations.getVinForReserv(credit_card)
130 
131         self.__vehicles.unreserveVehicle(vin)
132         self.__reservations.cancelReservation(credit_card)
133 
134     def calcRentalCost(self, vehicle_type, rental_period,
135                        want_insurance, miles_driving):
136         """Returns estimate of rental cost for provided parameter values.
137 
138            Returns dictionary with key values: {'base_charges', 'insur_rate',
139            'num_free_miles', 'per_mile_charge', 'estimated_mileage_charges'}
140         """
141 
142         return self.__vehicle_costs.calcRentalCost(vehicle_type, rental_period,
143                                                    want_insurance, miles_driving)
144 
145     # ---- Private Methods
146 
147     def __populateVehicles(self, vehicle_file):
148         """Gets vehicles from vehicle_file. Raises InvalidFileFormatError."""
149 
150         empty_str = ''
151 
152         # init vehicle string file headers
153         vehicle_file_headers = ('#CARS#', '#VANS#', '#TRUCKS#')
154         vehicle_type_index = 0
155 
156         # read first line of file (#CARS# expected)
157         vehicle_str = vehicle_file.readline()
158         vehicle_info = vehicle_str.rstrip().split(',')
159         file_header_found = vehicle_info[0]
160         expected_header = vehicle_file_headers[0]
161 
```

FIGURE 10-47 SystemInterface Class (*Continued*)

addReservation , and cancelReservation). Finally, method calcRentalCost (**lines 134–143**) returns the calculated rental charges for a given vehicle type, rental period, insurance option, and expected miles driven.

The remainder of the class consists of private supporting methods. Private method populateVehicles reads the information given by VEHICLES_FILENAME and populates the Vehicles instance. Private method populateCosts reads the information given by VEHICLE_COSTS_FILENAME and populates the VehicleCosts instance.

```
        if file_header_found != expected_header:
            raise InvalidFileFormatError(expected_header, VEHICLES_FILENAME)
        else:
            # read next line of file after #CARS# header line
            vehicle_str = vehicle_file.readline()

            while vehicle_str != empty_str:

                # convert comma-separated string into list of strings
                vehicle_info = vehicle_str.rstrip().split(',')

                if vehicle_info[0][0] == '#':
                    vehicle_type_index = vehicle_type_index + 1
                    file_header_found = vehicle_info[0]
                    expected_header = vehicle_file_headers[vehicle_type_index]

                    if file_header_found !=  expected_header:
                        raise InvalidFileFormatError(expected_header,
                                                     VEHICLES_FILENAME)
                else:

                    # create new vehicle object of the proper type
                    if file_header_found == '#CARS#':
                        vehicle = Car(*vehicle_info)
                    elif file_header_found == '#VANS#':
                        vehicle = Van(*vehicle_info)
                    elif file_header_found == '#TRUCKS#':
                        vehicle = Truck(*vehicle_info)

                    # add new vehicle to vehicles list
                    self.__vehicles.addVehicle(vehicle)

                # read next line of vehicle information
                vehicle_str = vehicle_file.readline()

    def __populateCosts(self, cost_file):
        """Populates RentalCost objects from provided file object."""

        # skip file header / read first line of file
        cost_file.readline()
        cost_str = cost_file.readline()

        for veh_type in VEHICLE_TYPES:
            # strip off newline (last) character and split into list
            cost_info = cost_str.rstrip().split(',')

            for cost_item in cost_info:
                cost_item = cost_item.strip()

            # add Vehicle Type/Rental Cost key/value to dictionary
            self.__vehicle_costs.addVehicleCost(veh_type,
                                                VehicleCost(*cost_info))

            # read next line of vehicle costs
            cost_str = cost_file.readline()
```

FIGURE 10-47 SystemInterface Class

We next look at the implementation of the final class, the RentalAgencyUI, given in Figure 10-48. Any user interface for the rental agency system must

access it through the system interface.
Thus, the __init__ method (**lines 14–16**) of the RentalAgencyUI class is passed a reference
to the system interface to store (in a private variable) providing it access to the system. The only

```
"""
    This module provides class RentalAgencyUI, a console user interface.

    Method start begins execution of the interface. Raises IOError exception.
"""
from systemInterface import VEHICLE_TYPES
from vehicles import InvalidVinError
from vehicleCosts import DAILY_RENTAL, WEEKLY_RENTAL, WEEKEND_RENTAL
from reservations import Reservation

class RentalAgencyUI:
    """This class provides a console interface for the rental agency system."""

    def __init__(self, sys):
        """Stores the provided reference to the vehicle rental system."""
        self.__sys = sys

    def start(self):
        """Begins the command loop."""

        self.__displayWelcomeScreen()
        self.__displayMenu()

        selection = self.__getSelection(7)

        while selection != 7:
            self.__executeCmd(selection)
            self.__displayMenu()
            selection = self.__getSelection(7)

        print('Thank you for using the Friendly Rental Agency')


    # Private Methods

    def __displayWelcomeScreen(self):
        """PRIVATE: Displays welcome message and general instructions."""

        print('*****************************************************')
        print('  * Welcome to the Friendly Vehicle Rental Agency *')
        print('*****************************************************')

    def __displayMenu(self):
        """PRIVATE: Displays a list of menu options on the screen."""

        print('\n<<< MAIN MENU >>>')
        print('1 - Display vehicle types')
        print('2 - Check rental costs')
        print('3 - Check available vehicles')
        print('4 - Get cost of specific rental')
        print('5 - Make a reservation')
        print('6 - Cancel a reservation')
        print('7 - Quit\n')

```

FIGURE 10-48 RentalAgencyUI Class (*Continued*)

other public method of the class is method start (**lines 18–31**). This method begins the command loop when called—that is, the repeated action of displaying the main menu, getting the user's selection, and executing the selected

command. The loop continues until a value of 7 (to quit the program) is entered. The rest of the methods of the class are private methods in support of the execution of commands in the start method.

```
    def __getSelection(self, num_selections, prompt='Enter: '):
        """PRIVATE: Returns user-entered value in range 1-num_selections."""

        valid_input = False

        selection = input(prompt)

        while not valid_input:
            try:
                selection = int(selection)

                if selection < 1 or selection > num_selections:
                    print('* Invalid Entry*\n')
                    selection = input(prompt)
                else:
                    valid_input = True
            except ValueError:
                selection = input(prompt)

        return selection

    def __displayDivLine(self, title = ''):
        """PRIVATE: Displays line of dashes, with optional title."""

        if len(title) != 0:
            title = ' ' + title + ' '
        print(title.center(70,'-'))

    def __executeCmd(self, selection):
        """PRIVATE: Executes command for provided menu selection."""

        if selection == 1:
            self.__CMD_DisplayVehicleTypes()
        elif selection == 2:
            self.__CMD_DisplayVehicleCosts()
        elif selection == 3:
            self.__CMD_PromptAndDisplayAvailVehicles()
        elif selection == 4:
            self.__CMD_DisplaySpecificRentalCost()
        elif selection == 5:
            self.__CMD_MakeReservation()
        elif selection == 6:
            self.__CMD_CancelReservation()

    def __CMD_DisplayVehicleTypes(self):
        """PRIVATE: Displays vehicle types (Cars, Vans, Trucks)."""

        self.__displayDivLine('Types of Vehicles Available for Rent')
        self.__displayVehicleTypes()
        self.__displayDivLine()
```

FIGURE 10-48 RentalAgencyUI Class (*Continued*)

The methods provided by the system interface correspond to the user options given in the main menu in the user interface (as shown in Figure 10-49).

Pri vate method displayWelcomeScreen (**lines 36–41**) provides the welcome message of the program. Private method displayMenu (**lines 43–53**) is repeatedly called to redisplay the users' options before each next command.

Private method getSelection is called whenever the user is to enter a number within a provided range, such as in the main menu. Such selection occurs in other places in the program as well. Therefore, the method is designed to be passed an argument indicating the upper limit of the range of selections. (For example, for the selection of

```
    def __CMD_DisplayVehicleCosts(self):
        """PRIVATE: Displays rental costs for Cars, Vans, Trucks."""

        empty_str = ''
       blank_char = ' '

       # get vehicle type from user
       self.__displayVehicleTypes()
       vehicle_type_num = self.__getSelection(len(VEHICLE_TYPES))
       vehicle_type = VEHICLE_TYPES[vehicle_type_num - 1]

       # set column headings
       row1_colheadings = blank_char * 24 + format('Free', '7') + \
                 format('Per Mile', '10') + format('Insurance', '17')

       row2_colheadings = format('Daily', '7') + format('Weekly', '8') + \
                 format('Weekend', '9') + format('Miles', '7') + \
                 format('Charge', '10') + format('(per day)', '17')

       # get vehicle costs
       costs = self.__sys.getVehicleCosts(vehicle_type)

       # build costs line to display
       costs_str = empty_str

       field_widths = ('7', '8', '9', '7', '10', '17')
       for k in range(0, len(costs)):
           costs_str = costs_str + format(costs[k],'<' + field_widths[k])

       # display headings and costs line
       self.__displayDivLine('Rental Charges for ' + vehicle_type + 's')
       print()
       print(row1_colheadings + '\n' + row2_colheadings + '\n' + costs_str)
       self.__displayDivLine()

    def __CMD_DisplayAvailVehicles(self, vehicle_type, numbered=False):
       """PRIVATE: Displays unreserved vehicles of selected type.
           When default argument is True, lists vehicles with sequential
           numbers at the start of the line of each vehicle description.
       """

       avail_vehicles_list = self.__sys.getAvailVehicles(vehicle_type)
       self.__displayDivLine('Available ' + vehicle_type + 's')

       if avail_vehicles_list == []:
           print('* No Vehicles Available of this Type *')
       elif numbered:
           k = 1
           for veh in avail_vehicles_list:
               print(str(k) + '-' + veh.getDescription())
               k = k + 1
       else:
           for veh in avail_vehicles_list:
               print(veh.getDescription())
```

FIGURE 10-48 RentalAgencyUI Class (*Continued*)

items from the main menu, getSelection would be passed the value 7.) Two types of errors are checked by the method. One is if a value is outside of the allowable

range. The other checks for invalid type of input (such as entering a letter instead of a number). The method requires that an appropriate value be entered by the user before returning.

```
    def __CMD_PromptAndDisplayAvailVehicles(self):
        """PRIVATE: Prompts user for vehicle type, and displays all
            available vehicles of that type.
        """

        self.__displayVehicleTypes()
        vehicle_type_num = self.__getSelection(len(VEHICLE_TYPES))
        vehicle_type = VEHICLE_TYPES[vehicle_type_num - 1]
        self.__CMD_DisplayAvailVehicles(vehicle_type)

    def __CMD_DisplaySpecificRentalCost(self):
        """PRIVATE: Prompts user for selections and displays rental cost."""

        # get vehicle type and rental period from user
        self.__displayVehicleTypes()
        vehicle_type_num = self.__getSelection(len(VEHICLE_TYPES))
        vehicle_type = VEHICLE_TYPES[vehicle_type_num - 1]

        # assign tuple, e.g. ('DAILY_RENTAL', '4')
        rental_period = self.__getRentalPeriod()

        # prompt user for optional insurance
        want_insurance = input('Would you like the insurance? (y/n): ')

        while want_insurance not in ('y', 'Y', 'n', 'N'):
            want_insurance = input('Would you like the insurance? (y/n): ')

        # convert to Boolean value
        want_insurance = want_insurance in ('y', 'Y')

        # prompt user for num of miles expected to drive
        print('\nNumber of miles expect to drive?')
        num_miles = self.__getSelection(1000, prompt = 'Miles: ')

        # calc base rental cost
        rental_cost = self.__sys.calcRentalCost(vehicle_type, rental_period,
                                                want_insurance, num_miles)
        # display estimated rental cost
        print()
        self.__displayDivLine('ESTIMATED ' + vehicle_type + ' RENTAL COST')

        if want_insurance:
            print('Insurance rate of', rental_cost['insur_rate'], 'per day')
        else:
            print('* You have opted out of insurance coverage *')

        if rental_period[0] == DAILY_RENTAL:
            print('\nDaily rental for', rental_period[1],
                  'days would be $', format(rental_cost['base_charges'], '.2f'))
        elif rental_period[0] == WEEKLY_RENTAL:
            print('\nWeekly rental for', rental_period[1],
                  'weeks would be $', format(rental_cost['base_charges'], '.2f'))
        elif rental_period[0] == WEEKEND_RENTAL:
            print('\nWeekend rental is $', rental_cost['base_charges'])

        est_total_cost = rental_cost['base_charges'] + \
                         rental_cost['estimated_mileage_charges']

        print('\nYour cost with an estimated mileage of',
              num_miles, 'miles would be', format(est_total_cost, '.2f'))

        print('which includes', rental_cost['num_free_miles'],
              'free miles and a charge of',
              format(rental_cost['per_mile_charge'], '.2f' ), 'per mile')

        self.__displayDivLine()
```

FIGURE 10-48 RentalAgencyUI Class (*Continued*)

```
def __CMD_MakeReservation(self):
    """PRIVATE: Prompts user for vehicle vin and reserves vehicle."""

    # get vehicle type
    self.__displayVehicleTypes()
    vehicle_type_num = self.__getSelection(len(VEHICLE_TYPES))
    vehicle_type = VEHICLE_TYPES[vehicle_type_num - 1]

    if self.__sys.numAvailVehicles(vehicle_type) == 0:
        print('Sorry - No available', \
              VEHICLE_TYPES[vehicle_type_num - 1] + 's', 'at the moment')
    else:
        self.__CMD_DisplayAvailVehicles(vehicle_type, numbered=True)
        avail_vehicles = self.__sys.getAvailVehicles(vehicle_type)

        valid_input = False

        while not valid_input:
            selected = input('\nEnter number of vehicle to reserve: ')

            if not selected.isdigit():
                print('Please enter the number preceeding the vehicle')
            elif int(selected) < 1 or \
                  int(selected) > self.__sys.numAvailVehicles(vehicle_type):
                print('\nINVALID SELECTION - Please re-enter: ')
            else:
                valid_input = True

        vin = avail_vehicles[int(selected) - 1].getVin()
        vehicle = self.__sys.getVehicle(vin)
        print(vehicle.getDescription())

        vehicle.setReserved(True)

        name = input('\nEnter first and last name: ')
        addr = input('Enter address: ')
        credit_card = input('Enter credit card number: ')

        reserv = Reservation(name, addr, credit_card, vin)
        self.__sys.addReservation(reserv)
        print('* Reservation Made *')

def __CMD_CancelReservation(self):
    """PRIVATE: Prompts user for credit card and cancels reservation."""

    credit_card = input('Please enter your credit card number: ')

    if self.__sys.findReservation(credit_card):
        print('Calling cancelReservation of sys')  # **************
        self.__sys.cancelReservation(credit_card)
        print('** RESERVATION CANCELLED **')
    else:
        print('* Reservation not Found - Invalid Credit Card Entered *')
```

FIGURE 10-48 RentalAgencyUI Class (*Continued*)

Method displayDivLine is called to display a row of dashes. It has a default parameter title. Thus, the method may be called with or without a supplied argument. If an argument is not supplied, then a complete row of dashes is displayed. If a title is displayed, it is centered within the displayed line of dashes as given below:

----------Types of Vehicles Available for Rent----------

```
def __displayVehicleTypes(self):
    """PRIVATE: Displays the numbered selection of vehicle types."""

    print('\nEnter type of vehicle')
    vehicle_types = self.__sys.getVehicleTypes()

    k = 1

    for veh_type in vehicle_types:
        print(k, '-', veh_type)
        k = k + 1

    print()
```

```
def __getRentalPeriod(self):
    """PRIVATE: Prompts user for desired rental period.

       Returns tuple of the form (x, n), where x is one of DAILY_RENTAL,
       WEEKLY_RENTAL or WEEKEND_RENTAL, and n is the number of those units,
       e.g. (DAILY_RENTAL, 3) three days of a daily rental.
    """
    print('\nEnter the rental period:')
    print(DAILY_RENTAL, '- Daily ', WEEKLY_RENTAL, '- Weekly ',
          WEEKEND_RENTAL, '- Weekend\n')

    valid_input = False

    while not valid_input:
        try:
            selection = int(input('Enter: '))

            while selection not in (DAILY_RENTAL, WEEKLY_RENTAL,
                                    WEEKEND_RENTAL):

                print('* Incorrect entry. Please Re-enter *\n')
                selection = int(input('Enter: '))

            if selection == DAILY_RENTAL:

                print('How many days do you need the vehicle?', end = '')
                   num_days = self.__getSelection(14, prompt = ' ')

                if num_days > 6:
                    print("Why don't you consider a weekly rental?\n")

                return (DAILY_RENTAL, num_days)
            elif selection == WEEKLY_RENTAL:
                print('How many weeks do you need the vehicle? (1-3): ')
                num_weeks = self.__getSelection(3, prompt = 'Weeks: ')

                return (selection, num_weeks)

            elif selection == WEEKEND_RENTAL:
                return (selection, 1)

        except ValueError:
            print('* Invalid Input - Number Expected *\n')
```

FIGURE 10-48 RentalAgencyUI Class

```
<<< MAIN MENU >>>                                  Methods of SystemInterface
1 - Display vehicle types          ---------->     getVehicleTypes()
2 - Check rental costs             ---------->     getVehicleCosts()
3 - Check available vehicles       ---------->     getAvailVehicles()
4 - Get cost of specific rental    ---------->     calcRentalCost()
5 - Make a reservation             ---------->     makeReservation()
6 - Cancel a reservation           ---------->     cancelReservation()
7 - Quit
```

FIGURE 10-49 Correspondence of Menu Selections and Methods of System Interface

Method executeCmd (**lines 83–97**) is called to execute each command of the main menu by selection number. Rather than place the code for each command within this method, a specific command method is called for each command. Private method CMD_DisplayVehicleTypes (**lines 99– 104**) displays the three types of vehicles (Cars, Vans, and Trucks). It in turn makes use of private method displayVehicleTypes (**lines 281–293**). Within this method, getVehicleTypes of the systemInterface is called to retrieve the types, since that is where they are defined.

Private method CMD_DisplayVehicleCosts (**lines 106–139**) displays the rental costs for a specific vehicle type:

```
Enter vehicle type: 1

------------- Following are the Rental Charges for Cars --------------

                        Free   Per Mile  Daily
Daily  Weekly  Weekend  Miles  Charge    Insurance
24.99  180.00  45.00    100    .15       14.99
----------------------------------------------------------------------
```

It displays a selection of vehicle types (**line 113**) for the user to choose from, constructs two rows of column headings (**lines 118–123**), and gets the vehicle costs for the selected vehicle type by call to method getVehicleCosts of the system interface (**line 126**). The rest of the method displays the information using formatted strings to align under the column headings.

Private method CMD_DisplayAvailVehicles displays all vehicles of a given vehicle type that are not currently reserved:

```
Enter vehicle type: 1
-------------------------- Available Cars --------------------------
Chevrolet Camaro     passengers: 4   doors: 2    mpg: 30   vin: WG8JM5492DY
Chevrolet Camaro     passengers: 4   doors: 2    mpg: 30   vin: KH4GM4564GD
Ford Fusion          passengers: 5   doors: 4    mpg: 34   vin: AB4FG5689GM
Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: GH2KL4278TK
Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: KU4EG3245RW
Chevrolet Impala     passengers: 6   doors: 4    mpg: 30   vin: QD4PK7394Jl
Chevrolet Impala     passengers: 6   doors: 4    mpg: 30   vin: RK3BM4256YH
```

It displays a selection of vehicle types (**lines 158–159**) for the user to select from, and gets the available vehicles for the selected type from the system interface by call to method getAvailVehicles of the system interface (**line 147**). The rest of the method simply displays, row by row, the information for each vehicle.

Pri vate method CMD__PromptAndDisplayAvailVehicles (**lines 161–169**) prompts the user for a vehicle type, and then displays the information of vehicles of that type not currently reserved. Private method CMD_DisplaySpecificRentalCost (**lines 171–226**) displays the costs of a particular vehicle rental based on the vehicle type, the time period rented, whether insurance is opted for, and the estimated number of miles driven:

```
--------------------- ESTIMATED Car RENTAL COST ----------------------
* You have opted out of the daily insurance *

Your cost of a daily rental would be 74.97

Your cost with an estimated mileage of 240 would be 95.97
which includes 100 free miles and a charge of 0.15 per mile
----------------------------------------------------------------------
```

First, it displays a selection of vehicle types for the user to choose from (**line 176**). It then requests the desired rental period by a call to private method getRentalPeriod (**line 180**). A tuple is returned containing constant value DAILY_RENTAL, WEEKLY_RENTAL, or WEEKEND_RENTAL, and the number of days or weeks rental period (with weekends defaulting to 1). The user is then asked if they want the optional insurance (**line 186**) and the number of miles expected to drive (**line 193**). The estimated rental cost is then calculated by a call to method calcRentalCost of the system interface. The rest of the method has to do with formatting and displaying the output.

Pri vate method makeReservation allows the user to reserve a vehicle of a given type (**line 227**). The method first checks if there are vehicles available (**line**

235). It then calls CMD__DisplayAvailVehicles (**line 239**) with the default argument set to True. This causes the vehicle descriptions to be listed as a number list so that the desired vehicle can be selected. On **lines 242–253**, the vehicle number selected is read, checking for any invalid selections. On **line 255**, the VIN is retrieved for the selected vehicle. The VIN is needed by method getVehicle (of the SystemInterface class) to retrieve a given Vehicle object. Having the object, it then calls method getDescription (**line 257**) to display the description of the vehicle to the user. It also sets the reservation status to True by a call to setReserved (**line 259**).

The user's name, address, and credit card number are requested (**lines 261–263**) to make the reservation. It does this by creating a new Reservation object constructed with the provided information (**line 265**) and adds it to the collection of reservations (**line 266**). Confirmation of the reservation is displayed (**line 267**). Private method CMD_CancelReservation (**line 269–279**) cancels reservations by the credit card number used for making the reservation. Finally, private methods displayVehicleTypes and getRentalPeriod, supporting methods of the command methods, are given on **lines 281–337**.

Putting it All Together

Figure 10-50 shows the main module of the Vehicle Rental Agency Program that puts everything together. It first creates the vehicle rental agency system (with a system interface) and then creates a user interface containing a reference to the system, providing the user with a natural means of accessing the system.

```
1  # Rental Agency Program
2
3  """ This program performs the tasks of a vehicle rental agency, including the
4      display of vehicle information and the ability to make/cancel reservations.
5  """
6  from systemInterface import SystemInterface
7  from rentalAgencyUI import RentalAgencyUI
8
9  try:
10     # create system with populated data from file
11     sys = SystemInterface()
12
13     # associate user interface with system
14     ui = RentalAgencyUI(sys)
15
16     # start the user interface
17     ui.start()
18
19 except IOError:
20     print('** PROGRAM TERMINATION (IO Error) **')
```

FIGURE 10-50 Rental Agency Program Main Module

The main module for the program imports the SystemInterface class (**line 6**) and the RentalAgencyUI class (**line 7**). First, an instance of the SytemInterface class is created (**line 11**). The __init__ method of the class is implemented to open the vehicle information file, as well as the vehicle cost file. The information in these files is used to populate the vehicle and vehicle cost objects in the system. If any file errors occur during this process, an IOError exception is raised by the system interface and caught (**line 19**) to terminate the program.

Once the files are successfully opened and read, an instance of the RentalAgencyUI class is created (**line 14**). It is passed the needed reference to the system interface. Once the user interface is created, the start method of the user interface is called to begin the command loop and receive and execute commands from the user.

CHAPTER SUMMARY General Topics Python-Specific Programming Topics

Classes
Encapsulation, Inheritance, and Polymorphism Public vs. Private Class Members
Getters and Setters
Name Mangling
Superclass (Base Class/Parent Class)
Subclass (Derived Class/Child Class)
Subclass vs. Subtype

UML (Unified Modeling Language)
Class Diagrams in UML
Association, Multiplicity, and Role Names in UML Composition vs. Aggregation
Defining Classes in Python
Denoting Public and Private Class Members in

Python
The Use of self in Python
Special Method __init__ in Python Arithmetic and Relational Special Methods in

Python
Inheritance and Polymorphism in Python Duck Typing in Python

CHAPTER EXERCISES
Section 10.1
1. What are the two kinds of entities “bundled” in a class?
2. What kind of entity can there be any number of instances created for a given class? **3.** What are the three fundamental features that object-oriented programming languages have in support of object-oriented programming?

Section 10.2

4. What else does encapsulation provide other than the ability to bundle together instance variables and methods?
5. Describe what it means for a member of a class to be private.
6. Explain the purpose of getters and setters.
7. Explain what the special identifier self is used for in Python.
8. Explain the use of name mangling in Python.
9. Explain when special methods __str__ and __repr__ are each used in Python.
10. Give an implementation of special method __str__ for a Range class (representing a range of integers) that contains integer instance variables, __start and __end, so that the value of Range objects are displayed as follows: '10 . . . 16', when output with print.
11. Give an implementation of special method __lt__ for the Range class of exercise 10 so that range1 , range2 evaluates to True if all the values in range1 are less than all the values in range2, and returns False otherwise.

Section 10.3
12. Give the one-line class definition header for a class named MySubclass that is a subclass of the class

MySuperclass .
13. Explain when a subclass can serve as a subtype.
14. For an object obj, show how in Python the type of the object may be determined from within a program or the Python shell.
15. Show how in the Python shell information about one of the built-in types of Python can be displayed.

Section 10.4
16. Explain the concept of polymorphism in object-oriented programming.
17. Explain the advantages of using polymorphism in program design.
18. What is meant by “duck typing” in Python?
Section 10.5
19. What does the name UML stand for?

20. What are the two types of diagrams in UML mentioned in the chapter, and what aspects of a program design does each represent?
21. Give a class diagram for the XYCoord class in the Let’s Try It box of section 10.2.2.
22. Give a class diagram that includes the partial description of the built-in str type in Figure 10-15, and the ExplodedStr subclass in Figure 10-16.
23. Give a class diagram for the Fraction class developed in section 10.2. Include the MixedFraction subclass in the diagram.

PYTHON PROGRAMMING EXERCISES

P1. Give a UML class diagram for a library. Include as entities tangible objects (such as books), persons (such as borrowers and librarians), and status (such as whether a book is checked out or not). Use multiplicity, navigation, and role names where appropriate.

P2. Design and implement a Money class that stores monetary values in dollars and cents. Special method __init__ should have the following function header,
def __init__(self, dollars, cents)

Include special method __repr__ (__str__) for displaying values in dollars and

cents: $ 0.45, $ 1.00, $ 1.25. Also include special method __add__, and three getter methods that each provide the monetary value in another currency. Choose any three currencies to convert to.

P3. Implement a class named AvgList as a subclass of the built-in list class in Python, able to compute the average of a list of numeric values. If the list contains any nonnumeric types, a ValueError exception should be raised.

P4. Design and implement a FootMeasure class that stores a linear measurement of feet and inches. Your class should have the following function header for special method __init__,
def __init__(self, feet 50, inches 50)
Thus, the class should be able to create a FootMeasure object in various ways by use of optional keyword arguments,

meas 5 FootMeasure()
meas 5 FootMeasure(feet 5 5)
meas 5 FootMeasure(feet 5 5, inches 58) meas 5 FootMeasure(inches 5 68)

Implement special method __repr__ in the class so that measurements are displayed as follows,
5 ft. NOT 5 ft. 0 in. 5 ft. 8 in. NOT 68 in.

When the measurement is 0, it should be displayed as, 0 ft. 0. ins. Include special method add() for adding FootMeasure values. Also include all the special methods for implementing the relational operators.

P5. Develop an abstract class named Temperature that stores a single temperature. The class should have the following function header for special method __init__,
def __init__(self, temperature)
The abstract class should contain the following methods:
__str__ — should return a string of the form “75 degrees Fahrenheit”

aboveFreezing() — returns True if temperature above the freezing point convertToFahren — returns a new Temperature object converted to degrees Fahrenheit convertToCelsius — returns a new Temperature object converted to degrees Celsius convertToKelvin — returns a new Temperature object converted to degrees Kelvin Develop the subclasses Fahrenheit, Celsius and Kelvin to appropriately implement each of the methods in the abstract Temperature class.

(Note that when a meaningless conversion method is applied, for example, temp1.convertToFahrenheit() where temp1 is an object of type Fahrenheit, then a copy of the Temperature object should be returned.)

Demonstrate the correctness of your classes by doing the following:
Temperature objects of a mix of Temperature types ♦ Create a list of

♦ Print out the value of each temperature in the list, and add “above freezing” if the temperature is above freezing (for the specific temperature scale).
♦ Create a new list of temperatures containing each temperature of the original list converted to a common temperature scale (Fahrenheit, Celsius, or Kelvin).
♦ For each temperature object in the new list, print out its temperature value, and if it is above the freezing point.

PROGRAM MODIFICATION PROBLEMS

M1. Fraction Class—Adding a Supporting Reduce Method
Complete the Fraction class in section 10.2.2 by implementing the missing methods using the specifications below,

```
def copy(self):
"""Creates a copy of a given Fraction."""

def reduce(self):
"""Reduces self to simplest terms. Also removes the signs if both numerator and
denominator are negative."""

def __adjust(self, factor):
"""Multiplies numerator and denominator by factor."""
```

M2. Vehicle Rental Agency Program—View Reservations
Modify the Vehicle Rental Agency program in section 10.6 to add the capability of viewing all current reservations.

M3. Division Operator for Fraction Class
Modify the Fraction class developed in section 10.2.2 to include a division operator by implementing special method __truediv__.

M4. Screen Display in Recipe Conversion Program
Modify the Recipe Conversion Program in section 10.2.3 so that the original

recipe ingredient measurements and the converted measurements are both displayed on the screen (in addition to having the converted recipe written to a file).

M5. Adjusted Measurements in the Recipe Conversion Program
Modify the Recipe Conversion Program in section 10.2.3 so that the units of measure for teaspoons and tablespoons in a converted recipe are displayed more appropriately. For example, if a recipe that calls for 2 teaspoons of baking soda is tripled, 6 teaspoons of baking soda are currently displayed in the converted recipe. Since there are 3 teaspoons to a tablespoon, the converted measurement would be more appropriately displayed as 2 tablespoons. Similarly, since there are 16 tablespoons in a cup, any number of tablespoons over 16 would be more appropriately displayed as one cup, plus some number of tablespoons.

M6. Revised Estimated Rental Cost in Vehicle Rental Agency Program
The rental cost for a particular type vehicle computed in the Vehicle Rental Agency Program is based on the number of days (or weeks) rented, the cost of optional insurance, and the estimated number of miles that the vehicle is expected to be driven. However, the estimate does not include the cost of gas that the customer must also pay. Since the miles per gallon (mpg) is available for all vehicles, modify the program so that this additional expense is added to the calculation of the estimated cost. The cost that is displayed should identify this additional expense. Use an instance variable in the VehicleCosts class to store the current price of gas per gallon.

M7. Additional Vehicle Type in Vehicle Rental Agency Program
Modify the Vehicle Rental Agency Program to include a fourth SUV vehicle type, in addition to cars, vans, and trucks. The attributes stored for an SUV object should include make and model, miles per gallon (mpg), number of passengers, whether or not it has automatic sliding doors, and the vehicle identification number (VIN). Make *all* changes in the program to incorporate this new vehicle type.

M8. Polymorphic Behavior of Vehicle Cost in Vehicle Rental Agency Program
Modify the Vehicle Rental Agency Program to include three subclasses of the VehicleCost class, CarCost, VanCost, and TruckCost so that VehicleCost can behave polymorphically. Make all changes to the program to incorporate these new vehicle cost types.

PROGRAM DEVELOPMENT PROBLEMS

D1. RGB/Hexadecimal Color Code Manipulation Program
Design an abstract class named ColorCode, and two subclass named RGBColorCode and HexColorCode. RGB color codes (as discussed in Chapter 9) are of the form (125, 80, 210), indicating the amount of red, green, and blue, respectively, for a given color. Each color value is on a scale of 0–255. Hexadecimal color codes are of the form F4F060, in which the first two hexadecimal digits indicate the amount of red, the middle two digits the amount of green, and the last two digits the amount of blue. Each pair of hexadecimal digits represent values in the range 0–255. Thus, these color encodings are just different representations of the same range of colors. Include in abstract class ColorCode methods for displaying a color code, for adding and reducing the amount of red, green, or blue by a given percentage, and for producing the complement of a given color. The complement is determined by subtracting each color value from 255. Develop a program in which the user can enter a series of colors in both RGB and hexadecimal form, and have them altered in the ways given above.

D2. Morse Code Translation Program
Implement a Morse code translator program from the UML design given below for translating English message files into Morse code, and Morse code files into English. The requirements for this program are as follows:

♦ English message files will be stored as text files, one sentence per line.

♦ Morse code message files will be stored as text files, one encoded character per line, where end of words are indicated by a blank line, and end of sentences by two blank lines.
♦ English message files will contain file extension .eng, and Morse code files file extension .mor. The assumptions for this program are as follows:

♦ Messages will only contain lowercase letters, the digits 0–9, periods, commas, and question marks.
Following is the UML diagram for the program.

The Translator class coordinates the translation of messages (English to Morse code, and Morse code to English). It contains two class members through composition: InputBuffer and OutputBuffer. Thus the Translator class is responsible for their construction.

The InputBuffer contains the current message line read from the file. If the file contains an English message, then an EnglishInputBuffer is used; if the files contains a Morse-coded message, then a MorseInputBuffer is used. As the Translator reads the current MesgChar from the InputBuffer, it requests the MesgChar to translate itself, sending it to the OutputBuffer. When isEndOfWord or isEndOfSentence is found true, it calls the corresponding MarkEndOfWord or MarkEndOfSentence method of the OuputBuffer. If the output is in English, a space character for each end of word and a period for each end of sentence is added to the buffer; if the output is in Morse code, then a blank line for each end of word, and two blank lines for each end of sentence is added to the buffer. This is repeated until the end of file has been reached.

Note that what a given MesgChar is depends on the type of message file reading. If the message being read is an English message, then a MesgChar is a single character (letter, digit, etc.). If, however, the message being read is a Morse-code message, then a MesgChar is a MorseChar (i.e., a string containing up to six dot and dash characters). Thus, when getting and putting MesgChars of a given InputBuffer and OutputBuffer, the number of characters actually retrieved or placed in the buffer depends on what type of buffer is involved. The Morse Code table for this assignment is given below.

Morse Code Table

A	.-	N	-.	1	.----	.	.-.-.-
B	-...	O	---	2	..---	,	--..--
C	-.-.	P	.--.	3	...--	?	..--..
D	-..	Q	--.-	4	....-	(	-.--.
E	.	R	.-.	5	.....	)	-.--.-
F	..-.	S	...	6	-....	-	-....-
G	--.	T	-	7	--...	"	.-..-.
H	....	U	..-	8	---..	_	..--.-
I	..	V	...-	9	----.	'	.----.
J	.---	W	.--	0	-----		
K	-.-	X	-..-				
L	.-..	Y	-.--				
M	--	Z	--..				

CHAPTER 11 Recursion

In Chapter 3, we covered the fundamental types of control—sequential, selection, and iterative—used to affect the control flow of programs. There is one final form of control that we have yet to cover, referred to as recursion. We look at the use of recursion and recursive problem solving in this chapter.

OBJECTIVES

After reading this chapter and completing the exercises, you will be able to: ♦ Describe the design of recursive functions

♦ Define infinite recursion

♦ Apply recursive program solving

♦ Explain the appropriate use of iteration vs. recursion ♦ Develop recursive functions in Python

CHAPTER CONTENTS

Motivation
Fundamental Concepts
11.1 Recursive Functions
11.2 Recursive Problem Solving 11.3 Iteration vs. Recursion Computational Problem Solving 11.4 Towers of Hanoi

MOTIVATION

Almost all computation involves the repetition of steps. Iterative control statements, such as the for and while statements, provide one means of controlling the repeated execution of instructions. Another way is by the use of *recursion*.

In *recursive problem solving*, a problem is repeatedly broken down into similar subproblems, until the subproblems can be directly solved without further breakdown. For example, consider the method of searching for a name in a sorted list of names below.

1. If the list contains only one name, then if the name found is the name you are looking for, then terminate with “name found,” otherwise terminate with “name not found.”

2. Otherwise, look at the middle item in the list. If that is the name you are looking for, then terminate with “name found.”

3. Otherwise, continue by searching in a similar manner either the top half of the list, if the name you are looking for is alphabetically before the middle name of the list, or the bottom half of list, if the name you are looking for is alphabetically after.

The first two steps of this method are straightforward. The detail comes when the list needs to be continually broken down into sublists, and the appropriate sublists are searched. The beauty of recursive problem solving, however, is that the details of how to solve (smaller) subproblems do not need to be specified—the same steps that were used on the original list still apply. Although it is natural to try to think through all the resulting steps that are taken to recursively

solve a problem, the power of “recursive thinking” is to understand that doing so is unnecessary. In this chapter, we demonstrate the power of recursive problem solving by looking at some classic examples that highlight its effectiveness.

FUNDAMENTAL CONCEPTS

11.1 Recursive Functions

Computational problem solving via the use of *recursion* is a powerful problem-solving approach. The development and use of recursive functions, however, requires a different perspective on computation than we have had so far. We discuss the design and use of recursive functions in this section.

11.1.1 What Is a Recursive Function?

A **recursive function** is often defined as “a function that calls itself.” While this is an accepted definition, it is not necessarily the most appropriate explanation, for it plants in one’s mind the image given in Figure 11-1.

FIGURE 11-1 Recursive Function Definition

The illustration in the figure depicts a function, A, that is defined at some point to call function A (itself). The notion of a self-referential function is inherently confusing. There are two types of entities related to any function however—the *function definition*, and any current *execution instances*.

What is meant by the phrase “a function that calls itself ” is a function *execution instance* that calls another *execution instance* of the same function. A function definition is a “cookie cutter” from which any number of execution instances can be created. Every time a call to a function is made, another execution instance of the function is created. Thus, while there is only one definition for any function,

there can be any number of execution instances. In order to fully understand the mechanism of recursive function calls, we first consider the general mechanism of non-recursive function calls as depicted in Figure 11-2.

FIGURE 11-2 General Function Calls

In the function calls in the figure, there is no trouble visualizing the sequence of events that occur. First, an execution instance from the definition of function A is created and begins executing. When the call to function B is reached, the execution instance of function A is suspended while an execution instance of function B is created and begins executing. In turn, when the function call to function C is reached, function B suspends execution while an execution instance of function C is created and begins executing.

This calling and suspending of executing function instances could (theoretically) continue indefinitely. However, in this case, function C does not make a call to any other function. Thus, it simply executes until termination, returning control to the function that called it, function B. Function B then continues its execution until terminating, returning control to the function that called it, function A. Finally, function A completes and terminates, returning control to wherever it was called from.

Now, let's consider the situation when the original function, function A, is a recursive function—that is, its definition includes a call to function A (itself). As depicted in Figure 11-3, each current execution instance of function A will spawn a *new* execution instance of function A.

FIGURE 11-3 Recursive Function Execution Instances

Note that the execution of a series of recursive function instances is similar to the execution of a series of non-recursive instances, except that the execution instances are "clones" of each other (that is, of the same function definition). Thus, since all instances are identical, the function calls occur in exactly the same place in each.

Clearly, if the definition of a recursive function were written so that the function calls itself unconditionally, then *every* execution instance would unconditionally call another execution instance, ad infinitum. Such a nonterminating sequence of calls is referred to as **infinite recursion**, similar to the notion of an infinite loop. Therefore, *properly designed recursive functions always conditionally call another execution instance* so that eventually the chain of function calls terminates.

Now that we have better understanding of recursive functions, we can use the description of a recursive function as "a function that calls itself," understanding that this means that the function *definition* is self-referential, while the function execution instances are not. Next we will look at a classic example of a recursive function, computing the factorial of a given number.

LET'S TRY IT From the Python Shell, enter the following and observe the results.

... def rfunc(n): ... def rfunc(n): print(n) if n 55 1: if n .0: return 1

rfunc(n 21) else: return n 1 rfunc(n 2 1) ... rfunc(4) ... rfunc(1)
??? ???
... rfunc(0) ... rfunc(3)
??? ???
... rfunc(100) ... rfunc(100)
??? ???

A **recursive function** is a function (definition) that conditionally calls itself.
11.1.2 The Factorial Function

We now look at a particular mathematical function, factorial, which is often defined by a recursive definition. We then look at how such a recursive definition can be written as a recursively defined program function.

The Recursive Definition of the Factorial Function
The factorial function is an often-used example of the use of recursion. The computation of the factorial of 4 is given as,
factorial(4) 5 4 ? 3 ? 2 ? 1 5 24
In general, the computation of the factorial of any (positive, nonzero) integer n is,
factorial(n) 5 n ? (n 2 1) ? (n 2 2) ? . . . 1

The one exception is the factorial of 0, defined to be 1. Note that if we apply this definition to the factorial of n 2 1, we get factorial(n 2 1) 5 (n 2 1)(n 2 2) . . . 1. Therefore, the factorial of n can be defined as n times the factorial of n 2 1,

factorial(n) 5 n ? (n 2 1) ? (n 2 2) ? . . . 1 factorial(n 2 1)
Thus, the complete definition of the factorial function is,
factorial(n) 51, if n 0 5 n ? factorial(n 2 1), otherwise

This definition of the factorial function is clearly defined in terms of itself, referred to as a **recursive definition**. The part of the definition “factorial(n) 5 1, if n 5 0” is referred to as the **base case**. The base case of a recursive definition is what terminates the repeated application of the definition (and thus the repeated function calls when executed).

Consider what would happen if the base case for the factorial function were not part of the definition,

factorial(n) 5 n ? factorial(n 2 1), for all n
Applying this definition, the computation of the factorial of 4 would be,
factorial(4) 5 4 ? 3 ? 2 ? 1 ? 0 ? 21 ? 22 ? 23 ? 24 ? . . .

The factorial of any (positive) number would be 0, since 0 would always be part of the product of values. Thus, not only is this an incorrect definition of the factorial function, if we implemented this definition as a recursive function, it would never terminate (and thus never produce a result).

Suppose, on the other hand, the base case for the definition of the factorial function is given,
but that n * factorial(n 2 1) were given as n * factorial(n 1 1) instead,
factorial(n) 51, if n 0 5 n ? factorial(n 1 1), otherwise
Applying this definition, the computation of the factorial of 4 would be,
factorial(4) 5 4 ? 5 ? 6 ? 7 ? 8 ? 9 ? 10 ? 11 ? 12 ? . . .

If we implemented this (incorrect) version of factorial as a recursive function, the function would also never terminate. The problem, however, it is not because a base case is not included. It is because *the problem is not being broken down into subproblems in which the base case can be applied*.

This highlights three important characteristics of any recursive function, given in Figure 11-4.

1. There must be at least one base case (a problem instance whose solution is known without further recursive breakdown).
2. Problems that are not a base case are broken down into subproblems that are a similar kind of problem as the original problem and work towards a base case.
3. There is a way to derive the solution of the original problem from the solutions of the recursively solved subproblems.

FIGURE 11-4 Requirements of a Properly Designed Recursive Function
Going back to the original (correct) definition of the factorial function therefore,
factorial(n) 5 51, if n 0
5 n ? factorial(n 2 1), otherwise

we see that the first condition holds since the base case, factorial(0) 51, can be applied without any future recursive breakdown of the problem. It follows the second condition since the problem is broken down into a subproblem that is a smaller instance of the original. Finally, it meets that third condition since the results of the original problem can be determined by multiplying the solution of

each subproblem. Thus, this is a properly defined recursive function. We next look at an actual implementation of a recursive factorial function.

Every properly defined recursive function must have at least one base case, and must redefine the problem into subproblems that work towards a base case such that the solution of the original problem can be derived from the solutions of the recursively solved subproblems.

A Recursive Factorial Function Implementation
Given a recursive definition of the factorial function, we can simply write it as Python program code. This is given in Figure 11-5.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

FIGURE 11-5 Recursive Factorial Function
Implementation

Examination of this function reveals that the recursive function call is conditionally made. That is, only if n is not equal to zero is another execution instance created, otherwise the current execution instance terminates and returns a value of 1. Termination is guaranteed since the initial value of parameter n is required to be greater than or equal to 0, and each next function call operates on smaller values (i.e., n 2 1). Finally, the solutions of all of the subproblems provide a solution to the original one. The sequence of function execution instances generated for the factorial of 4 is given in Figure 11-6.

FIGURE 11-6 Factorial Recursive Instance Calls

Each execution instance of function factorial is suspended while the evaluation of expression n * factorial(n 21) is completed. When factorial is finally called with the value 0, five execution instances of the function exist, the first four suspended until instance factorial(0) completes. When factorial(0) returns the value 1, the evaluation of expression 1 * factorial(0) can be completed, returning 1 as the value of factorial(1), and so on, until 4 * factorial(3) is evaluated and 24 is returned as the value of the original function call.

Finally, we point out that although the factorial function serves as a good example, in practice, recursion is an inappropriate choice for implementing this function. This is because an iterative version can be easily implemented, providing a much more efficient computation. (We discuss the appropriate use of recursion in section 11.3.)

LET'S TRY IT From the Python Shell, enter the following and observe the results. ... def factorial(n):
if n 55 0:
return 1
return n * factorial(n 2 1)

... factorial(4) ???
... factorial(0) ???
... factorial(100) ???
... factorial(10000) ???
... def ifactorial(n):

result 5 1
if n 55 0:
return result

for k in range(n, 0, 21): result 5 result * k
return result
... ifactorial(0)
???
... ifactorial(100)
???
... ifactorial(10000)
???

Although the factorial function is an often-used example of a recursive function, the function can be executed more efficiently when implemented to use iteration.

11.1.3 Let's Apply It—Fractals (Sierpinski Triangle)

Figure 11-7 illustrates a well-known recursive set of images called the Sierpinski triangle. The Sierpinski triangle is an example of a *fractal*. A fractal is a shape that contains parts that are similar to the whole shape, thus having the property of self-similarity (Figure 11-8). The turtle graphics program in Figure 11-9 generates Sierpinski triangles at various levels of repetition. This program utilizes the following programming features:

recursive functions➤

FIGURE 11-7 Sierpinski Triangle (Fractal Image)

FIGURE 11-8 Other Fractal Images

In addition to the turtle module, the program also imports the math module, needed for the square root function in the calculation of the height of a triangle (in function triangleHeight on **lines 21–25**). In the Sierpinski triangle, each next level in the pattern replaces each triangle with three smaller triangles. In order for the position of each next triangle to be determined (by functions getLeftTrianglePosition, getRightTrianglePosition, and getTopTrianglePosition at **lines 27**, **38**, and **49**, respectively) both the length of the sides of the triangle, as well as its height are needed. This is depicted in Figure 11-10.

The position of turtle shapes in turtle graphics is relative to the center of the shape. Each triangle is positioned relative to its center point (shown by the dot in the figure). Thus, method getLeftTrianglePosition calculates the location of the bottom left triangle as,

[position [0] 2 side / 4, position [1] 2 triangleHeight(side) / 4]

```
# Sierpinski Triangle Program

import turtle
import math

def createTriangleShape(coords):

    """Creates turtle shape from coords. Registers as 'my_triangle'. """

    turtle.penup()
    turtle.begin_poly()
    turtle.setposition(coords[0])
    turtle.setposition(coords[1])
    turtle.setposition(coords[2])
    turtle.setposition(coords[0])
    turtle.end_poly()

    tri_shape = turtle.get_poly()
    turtle.register_shape('my_triangle', tri_shape)

def triangleHeight(side):

    """Returns height of equilateral triangle with length side."""

    return math.sqrt(3) / 2 * side

def getLeftTrianglePosition(position, side):

    """Returns position of bottom left triangle in larger triangle.

       Returns (x,y) position for provided position and side length of
       larger triangle to be placed within.
    """

    return (position[0] - side / 4,
            position[1] - triangleHeight(side) / 4)

def getRightTrianglePosition(position, side):

    """Returns position of bottom right triangle in larger triangle.

       Returns (x,y) position for provided position and side length of
       larger triangle to be placed within.
    """

    return (position[0] + side / 4,  \
            position[1] - triangleHeight(side) / 4)
```

FIGURE 11-9 Sierpinski Triangle Program (*Continued*)

As a result, the x (horizontal) position of the lower left triangle is one quarter of the length of the side less than the larger triangle, therefore positioned to the left of the larger triangle's location. The y (vertical) position of the smaller triangle is one quarter of the height of the triangle less than the larger triangle, thus placed below the position of the original triangle. Positioning of the lower right triangle

(by method getRightTrianglePosition) is similarly determined (positioned to the right of and below the location of the original triangle). Finally, method getTopTrianglePosition determines the position of the smaller top triangle to be at the same x location as the original triangle, and one quarter of the height of the triangle higher.

```
def getTopTrianglePosition(position, side):

    """Returns x,y position of top triangle within larger one.

       For triangle at position, with length side.
    """

    return (position[0], position[1] + triangleHeight(side) / 4)

def drawSierpinskiTriangle(t, len_side, levels):

    """Recursive function that draws a Sierpinski triangle.

       Draws the number of levels of triangle given in levels.
    """

    if levels == 0:
        t.color('black')  # display triangle
        t.showturtle()
        t.stamp()

        return

    # resize triangle to half its size
    stretch_width, stretch_length, outline = t.turtlesize()
    t.turtlesize(0.5 * stretch_width, 0.5 * stretch_length, outline)

    # determine positions for each of the three embedded triangles
    left_triangle_position = getLeftTrianglePosition(t.position(),
                                                     len_side)
    right_triangle_position = getRightTrianglePosition(t.position(),
                                                       len_side)
    top_triangle_position = getTopTrianglePosition(t.position(),
                                                   len_side)

    # recursively display left triangle
    t.setposition(left_triangle_position)
    drawSierpinskiTriangle(t, len_side / 2, levels - 1)
    t.turtlesize(0.5 * stretch_width, 0.5 * stretch_length, outline)

    # recursively display right triangle
    t.setposition(right_triangle_position)
    drawSierpinskiTriangle(t, len_side / 2, levels - 1)
    t.turtlesize(0.5 * stretch_width, 0.5 * stretch_length, outline)

    # recursively display top triangle
    t.setposition(top_triangle_position)
    drawSierpinskiTriangle(t, len_side / 2, levels - 1)
    t.turtlesize(0.5 * stretch_width, 0.5 * stretch_length, outline)
```

FIGURE 11-9 Sierpinski Triangle Program (*Continued*)

Once the proper positioning of the smaller triangles is determined, the rest of the program is based on the use of recursion to repeatedly apply this division of triangles until a given number of levels have been drawn. Function createTriangleShape (**lines 6–19**) creates an equilateral triangle by positioning

the turtle to each of the screen locations provided in parameter coords. Because this movement occurs within the begin_poly and end_poly instructions, the polynomial

```
# ---- main

# set window size
turtle.setup(800, 600)

# get turtle
the_turtle = turtle.getturtle()

# init turtle
the_turtle.penup()
the_turtle.hideturtle()

# set the number of levels
num_levels = 3

# create triangle shape
coords = ((-240, -150), (240, -150), (0, 266))
createTriangleShape(coords)
len_side = 480

# create first triangle
the_turtle.shape('my_triangle')
the_turtle.setposition(0, -50)
the_turtle.setheading(90)

# call recursive function
drawSierpinskiTriangle(the_turtle, len_side, num_levels)
the_turtle.hideturtle()

# terminate program when close window
turtle.exitonclick()
```

FIGURE 11-9 Sierpinski Triangle Program

FIGURE 11-10 Relative Placement of Inner Triangles

drawn can be retrieved (**line 18**) and registered as a new turtle shape that can be

used within the program (**line 19**).

The main section of the program (**lines 101–129**) does the needed preparation before drawing can begin. On **line 102** the size of the turtle window is set. On **line 105** the (default) turtle is retrieved and named the_turtle. The turtle is then initialized so that the drawing capability is off (penup) and it is hidden. This is done since the only graphics to be produced by the turtle is when its (triangle) shape is stamped. Thus, the turtle is moved and resized to create all the stamped triangle images needed for creating a given Sierpinski triangle. On **line 112** the number of levels of the triangle is set. Thus, by changing this value, a Sierpinski triangle of various levels can be created.

On **line 115** a tuple of coordinates is defined that creates an equilateral triangle. (The absolute positions of these coordinates are not relevant, only their relative positions are used for defining the shape.) The length of the triangle is specified on **line 117**, matching the length of the triangle given by the specified coordinates. The turtle specified by the_turtle is set to shape 'my_triangle'. It is then positioned at location (0, 250) of the screen (a little below the center) and the heading is set to 90 degrees (to ensure that the triangle is pointing up).

Recursi ve function drawSierpinskiTriangle (**lines 58–97**) is passed three arguments: the turtle (in parameter t), the length of the sides of the overall triangle, and the number of levels of the Sierpinski triangle pattern to draw (**line 125**). As a recursive function, there must be a base case in which the function no longer calls itself. Since the number of levels starts at some nonzero value, the base case is reached when the value of parameter levels is 0 (**line 65**). At that point, the turtle size and location is specified for the smallest of the embedded triangles. Therefore, since no further breakdown of the triangles is needed, these lowest-level triangles are simply displayed.

When function drawSierpinskiTriangle is called with levels not equal to zero, the size of the current triangle shape of the turtle is cut in half (**lines 73–74**). The positions of each of the three smaller triangle to fit in the area of the current turtle shape are determined, and three recursive calls are made—one for each of the smaller triangles (**lines 85–97**). For each recursive call, the turtles are first positioned (**lines 85**, **90**, and **95**), and the recursive function calls made (**lines 86**, **91** and **95**). Because each recursive call resizes the turtle shape, on **lines 87**, **92**, and **97** the turtle shape is reset to what it was before each such call.

Self-Test Questions

1. A recursive function is best thought of as
(a) A function that calls itself
(b) A function execution instance that calls another execution instance of the same function

2. A recursive function call that never terminates is called . **3.** What does the following recursive function return as a value, for any given positive integer argument n?

```
def rfunc(n):
if n 55 0:
return 0
else:
return rfunc(n 2 1) 1 2
```

4. How many times does the factorial function in the chapter get called when computing the factorial of 4?
5. Iteration and recursion are two means of repeating a set of instructions. (TRUE/FALSE)
ANSWERS: 1. b, 2. infinite recursion, 3. 2n, 4. n 1 1, 5. True

11.2 Recursive Problem Solving

11.2.1 Thinking Recursively

We commonly solve problems by breaking a problem into subproblems, and separately solving each subproblem. Recursive problem solving is the same, except that a problem is broken down into subproblems that are another instance of the original type problem. This was true of the factorial function in section 11.1. The problem of solving the factorial of n was broken down into two subproblems—the multiplication by n as one subproblem, and the determination of the factorial of n 2 1 as the other, as shown in Figure 11-11.

The factorial function is an “easy” example of recursion, because the function itself is defined recursively. Thus, the recursive function developed is just an implementation of the mathematical definition. The use of recursion is most interesting when applied to problems that are not recursively defined. To do this, we need to meet the three requirements of recursive functions given earlier in Figure 11-4.

The power of recursion is that it provides a conceptually elegant means of

problem solving. The “elegance” derives from the fact that there is no need to specify or even think through all the steps that are taken to solve the problem. Since the recursive subproblems are the same kind of problem as the original, specifying the solution of the overall problem provides sufficient detail for solving each of the similar subproblems. To illustrate this, we look at some of the most well-known problems submitting to a recursive solution. We begin with an efficient means of sorting called MergeSort.

FIGURE 11-11
Subproblems of Factorial Function

The **power of recursion** is that it provides a conceptually elegant means of problem solving.

11.2.2 MergeSort Recursive Algorithm

In this section we look at how we can apply recursive problem solving to the problem of sorting lists. In order to solve this problem recursively, we have to imagine how the problem of sorting can be broken into subproblems, so that one or more of the subproblems is also the problem of sorting a (smaller) list.

When breaking a problem down, it is often most effective to break the problem into equal size subproblems. We consider, therefore, breaking down the problem of sorting n elements, into two subproblems of sorting lists of size n/2, as depicted in Figure 11-12.

FIGURE 11-12
Breakdown of the Problem of Sorting a List

At this point, the power of recursive thinking comes into play. Since we are developing a method for sorting lists of size n, *we can assume that any list of size less than n can be sorted, without needing to determine in detail how that is done*. (This is closely related to proof by induction in mathematics.) Thus, we can continue to develop our method for sorting lists of size n based on that fact.

The next step is to identify a base case that does not need to be broken down any further in order to be solved. The obvious base case is lists of size 1, since by definition they are sorted. Since we are dividing each list into two sublists at each step of the recursion, eventually each sublist will be of size 1. Thus, the breakdown into subproblems lead towards the base case, as required.

The last step is to determine how two (recursively solved) sorted sublists can be combined into one complete sorted list as depicted in Figure 11-13.

FIGURE 11-13 After Sorting Each of the Two Sublists

We cannot simply concatenate one sublist with the other. We need to somehow *merge* the values into one properly sorted list. This can be done as follows. Compare the smallest (top) item in each of the two lists. Whichever is the smaller value, move that as the first item of a new list. Cross off the item moved. Continue in the same manner until all the elements have been moved to the new list. The first steps of this process are shown in Figure 11-14.

FIGURE 11-14 The Process of Merging Two Sorted Lists

Now that we have determined how to break down the problem into (smaller) subproblems, and how to combine the solutions of the subproblem into a solution of the larger problem, we can specify the complete steps of this "merge sort" algorithm.

1. Split the list into two sublists.
2. Sort each sublist.
3. Merge the sorted sublists into one sorted list.

Figure 11-15 shows how this recursive method works on a particular list.

FIGURE 11-15 Recursive Problem-Solving Example (MergeSort)

The initial problem is to sort the list 12, 9, 4, 15, 18, 3. This is broken down into the problem of sorting the sublists 12, 9, 4 and 15, 18, 3. Since each sublist will be sorted in a similar manner, let's follow how the sublist 12, 9, 4 is sorted. It is

broken down into the two sublists 12, 9 and 4. Since the sublist containing only the value 4 is by definition sorted (as a base case) that subproblem does not need to be broken down any more—it is solved. So the sublist 12, 9 is broken down into the list containing 12 and another containing 9. These are now also solved since they are each a base case. The diagram indicates how the original sublist 15, 18, 3 is similarly broken down.

Now that since all base cases have been reached, the process of merging sublists begins as shown on the right side of the figure. Thus, to produce the sorted list 9, 12 the two base cases 9 and 12 are merged. We have shown how this merging can be systematically done. To produce the sublist 4, 9, 12, the *sorted* sublist 9, 12 is merged with the *sorted* sublist 4. The sorted sublist 3, 15, 18 is similarly produced. Thus, the final step is to merge the sorted versions of the original two subproblems, 4, 9, 12 and 3, 15, 18. We next give a Python implementation of this algorithm.

11.2.3 Let's Apply It—MergeSort Implementation

An implementation of the MergeSort algorithm is given in Figure 11-16. This program utilizes the following programming features.

➤ recursive functions

Function mergesort implements the recursive algorithm given above. When mergesort receives a list of length 1 (the base case), it simply returns the list (**lines 3–4**). Otherwise, the list is broken into two sublists, sublist1 and sublist2 (**lines 6–7**). Note that integer division is used, len(lst)//2, so that we don't compute non-integer lengths. Thus, when lst contains an uneven number of elements, for example 5, sublist1 and sublist2 will be of lengths 2 and 3, respectively, which makes no difference in the algorithm.

Once the list is divided into sublists, each of the sublists is sorted by a recursive call to mergesort (**lines 9–10**). Utilizing the power of recursive thinking, *we can assume that these recursive calls work without having to think through all the recursive steps*. Attempting to think

```
1  def mergesort(lst):
2
3      if len(lst) == 1:
4          return lst
5
6      sublist1 = lst[0:len(lst) // 2]
7      sublist2 = lst[len(lst) // 2:len(lst)]
8
9      sorted_sublist1 = mergesort(sublist1)
10     sorted_sublist2 = mergesort(sublist2)
11
12     return merge(sorted_sublist1, sorted_sublist2)
13
14
15 def merge(lst1, lst2):
16
17     merged_list = []
18
19     i = 0
20     k = 0
21
22     while i != len(lst1) and k != len(lst2):
23         if lst1[i] < lst2[k]:
24             merged_list.append(lst1[i])
25             i = i + 1
26         else:
27             merged_list.append(lst2[k])
28             k = k + 1
29
30     if i < len(lst1):
31         for loc in range(i, len(lst1)):
32             merged_list.append(lst1[loc])
33     elif k < len(lst2):
34         for loc in range(k, len(lst2)):
35             merged_list.append(lst2[loc])
36
37     return merged_list
```

FIGURE 11-16 Recursive Function mergesort

through the steps at each recursive level is tedious, confusing and unnecessary. Finally, the two sorted sublists are merged and returned as the final sorted list (**line 12**).

Note that supporting function merge is longer and more detailed than function mergesort. This should not be surprising, since the real work lies in the merging of sublists. In the function, the new merged list to be returned is constructed in variable merged_list. Thus, merged_list is initialized as an empty list (**line 17**). Variables i and k are used to keep track of the position of the current elements being compared in each of sublist1 and sublist2; thus, each is initialized to 0 (**lines 19–20**).

The while loop in **lines 22–28** merges one more element into merged_list for each iteration of the loop. If the current item in lst1 (determined by the current value of i) is less than the current item in lst2 (determined by the current value of k), then the current item of lst1, lst1[i], is appended to merged_list. Otherwise, the current item of lst2, lst2[k], is appended. Once the while loop terminates, all of the elements of lst1 and/or lst2 have been merged. What is left to do is append any remaining items of either sublist whose elements have not yet been merged (when the two sublists are not of equal length). This is taken care of in **lines 30–35**. Finally, on **line 37**, the newly created merged list is returned.

We note that the MergeSort function makes two recursive calls. Such recursive functions are called *doubly recursive*. Double recursion makes the conversion of a recursive solution to an iterative one more difficult than a "singly recursive" algorithm. We shall see another example of a doubly recursive function in the Computational Problem Solving section of the chapter.

Self-T est Questions
1. Only problems that are recursively defined can be solved using recursion. (TRUE/FALSE)
2. The power of recursion is the execution speed over iteration. (TRUE/FALSE)

3. The base case in the MergeSort algorithm is,
(a) an empty list **(b)** a list of length one **(c)** a list of length two **4.** The MergeSort algorithm only works on lists of length two or more. (TRUE/FALSE) **5.** The MergeSort algorithm divides a list into equal, or nearly equal, sublists in each recursive level (except for the base case). (TRUE/FALSE)
ANSWERS: 1. False, 2. False, 3. (b), 4. False, 5. True

11.3 Iteration vs. Recursion

Recursion is fundamentally a means of repeatedly executing a set of instructions. The set of instructions are those of a function, and the repetition comes from the fact that the function is repeatedly executed, once for each recursive function call. Thus, recursion and iteration are two means of accomplishing the same result. Whatever can be computed using recursion can also be computed using iteration, and vice versa.

Since iteration and recursion are equivalent in terms of what can be computed, a natural question is "When should I use recursion, and when should I use iteration?" There is no clear-cut answer to this question, but there are some guidelines. Generally, a recursive function generally takes more time to execute

than an equivalent iterative approach. This is because the multiple function calls are relatively time-consuming. In contrast, while and for loops execute very efficiently. Thus, *when a problem can be solved both recursively and iteratively with similar programming effort, it is generally best to use an iterative approach*. Recall that the Factorial function was an example of such a situation. The iterative version of computing factorial is simple to implement, and executes much more efficiently.

On the other hand, some problems are very difficult to solve iteratively, and almost trivial to solve recursively. This is when recursion is most effectively used. A classic example of such a problem is the Towers of Hanoi, discussed next.

Whatever can be computed using recursion can also be computed using iteration, and vice versa.

COMPUTATIONAL PROBLEM SOLVING

11.4 Towers of Hanoi

In this section we look at a classic example of recursive problem solving in computer science, the Towers of Hanoi.

11.4.1 The Problem

The Towers of Hanoi problem (Figure 11-17) is based on a legend of unknown origin. According to the legend, there is a Vietnamese temple with a large room containing three pegs and 64 golden disks. Each disk has a hole in it so that it can be slipped onto any of the pegs. In addition, each disk is of different size. The 64 disks are moved by priests from one peg to another, with the following conditions,

FIGURE 11-17 Towers of Hanoi

♦ Only one disk can be moved at a time. ♦ At no time can a larger disk be placed on top of a smaller one.

When the complete pile of 64 disks have been moved, the world is to end. There is not much need for concern, however—assuming that the priests moved one disk per second, it would take roughly 585 billion years to finish!

11.4.2 Problem Analysis

We will first attempt to solve this problem for three disks to gain some insight into the problem, and then develop a general solution for any number of disks. Thus, we will solve the simple problem of moving three disks from peg A to peg C as shown in Figure 11-18.

FIGURE 11-18 Towers of Hanoi Problem for Three Disks

First, let's solve this the hard way, by considering every step that must be taken. The obvious first move is to remove the smallest disk, and place it on either peg B or peg C. If we place the smallest disk on peg C, then we must place the next smallest disk on peg B, resulting in the configuration in Figure 11-19.

We then consider our third move. We can place the disk currently on peg B back on peg A, but that will be undoing what we just did in the last step. So in order to make progress, we move the smallest disk on peg C somewhere. We can move it on either peg A or peg B, since in each case it would be placed on a larger disk (thus not violating the problem's conditions). Let's assume that we

FIGURE 11-19 Towers of Hanoi after Two Moves

move the smallest disk currently on peg C on top of the second smallest disk on peg B. This move results in the configuration shown in Figure 11-20.

FIGURE 11-20 Towers of Hanoi after Three Moves

Now, we can move the largest disk currently on peg A to peg C (since peg C is currently empty). Then we can move the smallest disk from peg B to peg A, then move the second smallest disk from peg B to peg C, and finally move the smallest disk from peg A to peg C, thereby solving the problem as shown in Figure 11-21.

FIGURE 11-21 Towers of Hanoi Final Moves

The question is, what have we learned by thinking through the solution of this problem for three disks? What insight have we gained into the problem involving four disks? Five disks? Attempting to think through the individual steps for a larger and larger number of disks quickly becomes overwhelming.

As you should expect, there is a much more elegant solution for this problem involving recursion. The fundamental steps of a recursive solution for the Towers of Hanoi problem is given below:

Step 1 : View the stack as two stacks, one on top of the other. Call the top stack Stack1, and the bottom stack Stack2.
Step 2: Recursively move Stack1 from peg A to peg B.
Step 3: Recursively move (the exposed) Stack2 from peg A to peg C.

Step 4: Recursively move Stack1 from peg B to peg C.

As we have said, in recursive problem solving, we do not need to specify the detailed steps solving any subproblem that is a smaller, similar problem to the original problem. Here, steps 2, 3, and 4 have to do with solving subproblems that are the same type of problem as the original—moving a stack of disks from one peg to another. It does not matter that the stacks being moved are different, or are being moved between different pegs. Each of the subproblems can be recursively solved in a similar way, and thus we can assume that they can be solved without explicitly specifying how. Therefore, in order to effectively solve the general problem, we only need to decide and specify the details of step 1, how to break down a stack of disks into two separate (sub)stacks. Let's consider different ways of breaking down this problem.

Suppose we start with the obvious breakdown, that is, dividing the stack of disks (six in this example) into two equal (or close to equal) size stacks. If we choose that approach, then after (recursively) moving the top half of the original stack to peg B and (recursively) moving the bottom half of the stack to peg C, we have the configuration given in Figure 11-22.

FIGURE 11-22 One Possible Breakdown of the Towers of Hanoi

Here is where we can benefit from recursive thinking. From this configuration, we need to (recursively) move the stack of disks on peg B to peg C to complete the problem solution. However, we are not allowed to move the whole stack at once, but instead must move each disk one at a time. This is accomplished by solving this problem in an identical way to the solution of the larger problem involving all disks. Without thinking through all of the details of each step, we know that the top disk on peg B must be moved at this point (since only top disks can be moved, and we do not want to move the disks on peg C, since they are currently in the correct position). So we can move the top disk from peg B to

either currently empty peg A, or on top of the disks on peg C.

Note that this is the smallest disk of all disks. *Therefore, once we place this smallest disk on either peg, that peg cannot be used to place any other disks on top of it*. But the one insight we should have gained by working through the detailed solution of three disks is that there is always the need for a spare peg when moving a stack of disks from one peg to another. Thus, once the smallest disk is placed on a peg, that peg becomes unusable for placing any other disks on it (and thus unusable as a spare peg), as shown in Figure 11-23 for peg A.

FIGURE 11-23 Inappropriate Breakdown of the Towers of Hanoi

Since we have determined that dividing the stack of disks into two (essentially) equal-size stacks will not work, we need to consider alternate methods of dividing the stack of disks. Suppose we take the opposite approach, and divide the stack into very *unequal*-sized stacks? That is, let's consider dividing the stack so that the top stack consists of the top (smallest) disk only, and the bottom stack consists of the rest of the disks. Let's see how this plays out.

Moving the top stack from peg A to peg B means moving the one smallest disk to peg B, as shown in Figure 11-24.

FIGURE 11-24 Another Possible Breakdown of the Towers of Hanoi

That is easy, since it is a stack of only one! Now, we must (recursively) move the

bottom stack (with all the remaining disks) from peg A to peg C. We know that this cannot be done in one step, and must be done by moving one disk at a time. We also know that in order to move such a stack of disks from any peg to any other peg, we must have a spare peg. However, since the smallest disk has been placed on peg B (our spare peg), that is unavailable for use! Again, then, this approach will not work!

Finally, we consider the correct approach. We again break down the stack of disks into very unequal-sized stacks. However, this time, we break it down so that the top stack consists of *all disks except the bottom (largest) disk*. Therefore, the correct approach would be to recursively move the top stack (of n 2 1 disks) from peg A to peg B. Then, simply move the largest (remaining) disk from peg A to peg C. Then recursively move the stack of disks from peg B to peg C (on top of the largest disk), as shown in Figure 11-25.

1_3
2
AB C FIGURE 11-25 Correct Breakdown of the Towers of Hanoi

Note that in this case, when moving the stack of disks from peg B to peg C, we have the needed spare peg, since peg A is currently empty, and peg C has on it the largest disk, making it possible to use either as a spare peg. Thus, our recursive solution is:

Step 1: Recursively move the top n 2 1 disks from peg A to peg B. Step 2: Move the one (remaining) largest disk from peg A to peg C. Step 3: Recursively move the n 2 1 disks from peg B to peg C.

As with MergeSort, this is a doubly recursive algorithm. This is a well-designed recursive solution because:
♦ *There is a base case*. When the stack of disks consists of only one disk, such a "stack" can be moved without further breakdown.
♦ *The recursive steps work towards the base case*. The subproblem involves the solution of smaller and smaller sized stacks of disks.
♦ *The solutions of the subproblems provide a solution for the original problem.* Solutions of subproblems eventually result in all the disks being on the destination peg in the required order.
11.4.3 Program Design and Implementation
Now that we have defined a recursive solution to this problem, implementation

of a program that solves it is surprisingly simple, given in Figure 11-26.

```
def towers(start_peg, dest_peg, spare_peg, num_disks):

    if num_disks == 1:
        print 'move disk from peg', start_peg, 'to peg', dest_peg
    else:
        towers(start_peg, spare_peg, dest_peg, num_disks - 1)
        print 'move disk from peg', start_peg, 'to', dest_peg
        towers(spare_peg, dest_peg, start_peg, num_disks - 1)
```

FIGURE 11-26 Recursive Function Implementation of the Towers of Hanoi

Hopefully you have gained an appreciation for the power and elegance of recursion through the examples given, and that you will be able to begin to apply “recursive thinking” in your own problem solving. We give a program utilizing turtle graphics that produces an animation of the solution of the Towers of Hanoi for a given number of disks in Figure 11-27.

In addition to the turtle module, the program also imports the time module (**lines 3 and 4**). The time module contains a sleep function used to control the speed at which the animation proceeds (**line 217**), pausing 0.5 seconds between the movement of disks.

The main module is in **lines 229–277**. On **line 230** a welcome message is displayed indicating what the program does. The call to setup on **line 233** establishes the turtle screen size (in pixels). The three pegs are positioned at the bottom center of the turtle screen (**line 236**) and the vertical separation of the disks on each peg is set to 24 pixels (**line 237**). (The separation between disks on a peg will be proportionally reduced in function calcDiskLocations so that smaller disks at the top of a stack of disks are closer together than larger disks.)

V ariables peg_A, peg_B, and peg_C are initialized to integer values 0, 1, and 2, respectively. Thus, these values can serve as index values for lists holding both the disk objects on each peg, and a list storing all disk locations. The number of disks to solve the problem for is retrieved from the user by a call to getNumDisks and assigned to variable num_disks (**line 245**). Function displayPegs is then called to display the three pegs (**line 249**) for the locations in variable peg_locations. In **lines 252–253** the image file names the required number of disks (specified in num_disks) are assigned to variable disk_images by a call to function getDisk Images. The image file names are then passed to

function registerDiskImages to be registered in turtle graphics as shapes to which turtle objects can be assigned.

The locations on each peg for where all the disks are to be placed is calculated by function calcDiskLocations (**lines 98–126**), called on **line 256**. The required number of disks

```
# Towers of Hanoi Program

from turtle import *
import time

def getNumDisks():

    """Returns number of disks requested by user."""
    num_disks = int(input('Enter the number of disks(2-8): '))

    while num_disks < 2 or num_disks > 8:
        print('Must select between 2 and 8 disks')
        num_disks = int(input('Enter the number of disks (2-8): '))

    return num_disks

def displayPegs(peg_locs):

    """Displays peg images at peg_locs."""
    # retrieve and register peg image
    peg_image = 'Peg.gif'
    register_shape(peg_image)

    # set vertical offset
    offset = 65

    # create temp peg-shaped turtle
    temp = Turtle()
    temp.penup()
    temp.shape(peg_image)

    # stamp three images of peg-shaped turtle
    for loc in peg_locs:
        temp.setposition(loc[0], loc[1] + offset)
        temp.stamp()

def getDiskImages(num):

    """Returns a list of disk image file names for num requested.

       Disk images in order from largest to smallest image.
    """
    # init empty images list
    images = []

    # init first image file name
    disk_image = 'Disk-1.gif'

    # create all disk images
    for k in range(0,num):
        images = ['Disk-' + str(k+1) + '.gif'] + images

    return images
```

FIGURE 11-27 Towers of Hanoi Program (*Continued*)

(turtle objects) are created by function createDisks (**lines 80–96**) called on

line 261. All disks are initially placed on the start peg (peg A) by function placeDisksOnStartPeg (**lines 128– 132**), called on **line 264**. After being properly placed, the disks are made visible by function makeDisksVisible (**lines 134–138**) called on **line 267**. Once the disks are placed, there is a

```
def registerDiskImages(images):

    """Registers disk image file names provided in images."""
    for image in images:
        register_shape(image)

def newPeg(image_file):

    """Returns a new turtle with peg shape in image_file."""
    peg = Turtle()
    peg.hideturtle()
    peg.shape(image_file)

    return peg

def newDisk(image_file):

    """Returns a new turtle with disk shape in image_file."""
    disk = Turtle()
    disk.penup()
    disk.hideturtle()
    disk.shape(image_file)

    return disk

def createDisks(images):

    """Returns a list of disk shape turtles for each image in images.

       Disks ordered by image size from largest to smallest.
    """
    # init empty list of disks for all three pegs
    disks = [[], [], []]

    # create disks with all disks on first peg
    for k in range(0, len(images)):

        disks[0].append(newDisk(images[k]))
        disks[1].append(None)
        disks[2].append(None)

    return disks

def calcDiskLocations(num_disks, peg_locs, separation):

    """Returns the calculated locations of all disks for all three pegs.

       Locations in order from largest (bottom) disk to smallest, with
       decreasing separation for smaller disks.
    """
```

FIGURE 11-27 Towers of Hanoi Program (*Continued*)

two-second delay by a call to the sleep function of the time module (**line 270**), followed by a call to the recursive function towers to begin the problem-solving process (**line 273**).

Function towers (**lines 211–225**) implements the recursive algorithm for solving

the problem. When called, it is passed the peg to be treated as the start peg (from which disks are to be moved), a destination peg (where the disks are to be moved), and a spare peg (used as temporary storage while moving disks), and the number of disks to move. The base case is when there is only

```
    # init
    saved_separation = separation
    all_disk_locations = []

    # caculate locations of disks for all three pegs
    for peg in range(0, 3):

        disk_locs = []

        for k in range(0, num_disks):

            xloc = peg_locs[peg][0]
            yloc = int(peg_locs[peg][1] + k * separation)

            disk_locs.append((xloc, yloc))
            separation = separation * 0.95

        all_disk_locations.append(disk_locs)
        separation = saved_separation

    return all_disk_locations

def placeDisksOnStartPeg(start_peg, disks, locations):

    """Places and displays disks on start peg in turtle graphics."""
    for k in (range(0, len(disks[start_peg]))):
        disks[start_peg][k].setposition(locations[start_peg][k])

def makeDisksVisible(start_peg, disks):

    """Makes visible all disk-shaped turtles on start_peg."""
    for k in range(0,len(disks[start_peg])):
        disks[start_peg][k].showturtle()

def removeTopDisk(peg, disks):

    """Returns the top disk of disks currently on peg."""
    # init k to top-most position of disks on peg
    k = len(disks[peg]) - 1
    found = False

    # find top-most position with disk in place
    while k >= 0 and not found:

        if disks[peg][k] != None:
             disk = disks[peg][k]
             disks[peg][k] = None
             found = True
        else:
            k = k - 1

    return disk
```

FIGURE 11-27 Towers of Hanoi Program (*Continued*)

one disk to move (checked for on **line 215**). In this case, the disk is simply moved from the start peg to the destination peg by call to moveDisk. Otherwise,

the top num_disks – 1 are (recursively) moved from the start peg to the spare peg, the single remaining (bottom) disk is then moved from the start peg to the destination peg, and the num_disks 21 disks on the spare peg are (recursively) moved to the destination peg.

```
159 def placeTopDisk(disk, peg, disks, disk_locations):
160
161     """Places disk on top of disks currently on peg."""
162     # init k to bottom-most position of disks on peg
163     k = 0
164     found = False
165
166     # assign disk to bottom-most position of peg with no disk
167     while k < len(disk_locations[peg]) and not found:
168
169         if disks[peg][k] == None:
170
171             disks[peg][k] = disk
172             found = True
173         else:
174             k = k + 1
175
176     return disk
177
178 def getNextAvailDiskLoc(peg, disks, disk_locations):
179
180     """Returns location for next disk on peg."""
181     # init k to bottom-most position of disks on peg
182     k = 0
183     found = False
184
185     # get bottom-most location of peg with no disk
186     while k < len(disks[peg]) and not found:
187
188         if disks[peg][k] == None:
189             loc = disk_locations[peg][k]
190             found = True
191         else:
192             k = k + 1
193
194     return loc
195
196 def moveDisk(from_peg, to_peg, disks, disk_locations):
197
198     """Moves disk on top of from_peg to top of to_peg."""
199     # get disk on top of from_peg
200     disk = removeTopDisk(from_peg, disks)
201
202     # get available loc for new disk on to_peg
202     new_loc = getNextAvailDiskLoc(to_peg, disks, disk_locations)
204
205     # move disk image
206     disk.goto(new_loc)
207
208     # add disk to to_peg list of disks
209     placeTopDisk(disk, to_peg, disks, disk_locations)
210
```

```
def towers(start_peg, dest_peg, spare_peg, num_disks, disks,
            disk_locations):

    """Begins the problem solving process."""
    if num_disks == 1:
        moveDisk(start_peg, dest_peg, disks, disk_locations)
        time.sleep(.5)

    else:
        towers(start_peg, spare_peg, dest_peg, num_disks - 1,
               disks, disk_locations)

        moveDisk(start_peg, dest_peg, disks, disk_locations)
        towers(spare_peg, dest_peg, start_peg, num_disks - 1,
               disks, disk_locations)

# ---- main

# program welcome
print('This program solves the Towers of Hanoi for up to eight disks')

# init turtle screen
setup(width=800, height=600)

# init display parameters
peg_locations = ((-150, -50), (0,-50), (150,-50))
disk_separation = 24

# initpeg numbers
peg_A = 0
peg_B = 1
peg_C = 2

# get number of disks
num_disks = getNumDisks()
print()

# display pegs
displayPegs(peg_locations)

# create and register disk images
disk_images = getDiskImages(num_disks)
registerDiskImages(disk_images)

# calc locations for all disks
disk_locs = calcDiskLocations(num_disks, peg_locations,
disk_separation)
```

```
# create and position number of disks requested
disks = createDisks(disk_images)

# place all disks on start peg
placeDisksOnStartPeg(peg_A, disks, disk_locs)

# make all disks visible
makeDisksVisible(peg_A, disks)

# delay
time.sleep(2)

# start movement of disks
towers(peg_A, peg_C, peg_B, num_disks, disks, disk_locs)

# exit turtle screen
print('\nDone - Click screen to quit')
exitonclick()
```

FIGURE 11-27 Towers of Hanoi Program

CHAPTER SUMMARY General Topics Python-Specific Programming Topics

Recursi ve Functions
Infinite Recursion
Factorial Function
Recursive Problem Solving MergeSort
Iteration vs. Recursion Fractals
Towers of Hanoi

CHAPTER EXERCISES Section 11.1
Recursive Functions in Python

1. For the following recursive functions, indicate which of the requirements of a properly designed recursive function each violates.
(a) def rfunc1(n):

return n 1 rfunc(n 2 1)
(b) def rfunc2(n):
if n 55 0:
return 1
return n 1 rfunc2(n 1 1)
2. For the following Sierpinski triangle, indicate how many times function drawSierpinskiTriangle is called in the Sierpinski triangle program of Figure 11-9.

Section 11.2

3. Give all the steps for MergeSort in sorting the following list.
11 4 16 9 5 12 2 14
4. Indicate the results produced by the recursive function MergeSort in Figure 11-16 if line 3 was replaced with, if len(lst) 55 0:

PYTHON PROGRAMMING EXERCISES
P1. Write both a nonrecursive and recursive function that determines if a given number is even or not.

P2. Write both a nonrecursive and recursive function that determines how many times a given letter occurs in a provided string.
P3. Write both a nonrecursive and recursive function that displays a provided string backwards.
P4. Write both a nonrecursive and recursive function that converts numbers to base 2.
P5. Write both a nonrecursive and recursive function that calculates the Fibonacci number for any positive integer, defined as follows,
fib(0) 5 0,
fib(1) 5 1,
fib(n) 5 fib(n 2 1) 1 fib(n 2 2)

P6. Write both a nonrecursive and recursive function that displays the rows of asterisks given below, **********
****
****
******

****
**

P7. Write both a nonrecursive and recursive function that displays the rows of asterisks given below, **
****
******
****
****
****

Program Modification Problems 489
PROGRAM MODIFICATION PROBLEMS

M1. Towers of Hanoi: Reversal of Pegs
Modify the Towers of Hanoi program in section 11.4.3 so that the initial stack of disks is placed on peg C (as the start peg), and the final stack of disks is placed on peg A (the destination peg).

M2. Towers of Hanoi: Displayed List of Moves
Modify the Towers of Hanoi program in section 11.4.3 so that in addition to the movement of disks on the screen, the list of moves is displayed in the Python shell as given below,

Move disk from peg A to peg C
Move disk from peg A to peg B
Move disk from Peg C to peg A
etc.

M3. Sierpinski Triangle Program: Multiple Levels of Fractal Displayed
Modify the Sierpinski Triangle Program in section 11.1.3 so that it displays fours levels of the fractal as given below on the screen at once.

M4. Sierpinski Triangle Program: Modified for Creating Sierpinki Carpet
Modify the Sierpinski Triangle Program in section 11.1.3 so that it instead displays a Sierpinski carpet as the repeated pattern of a solid square surrounded by eight smaller squares as depicted below.

M5. Sierpinski Triangle Program: Modified for a Koch Snowflake
Modify the Sierpinski Triangle Program in section 11.1.3 so that a Koch snowflake fractal is generated instead. The progressive levels of a Koch snowflake fractal, like the Sierpinski triangle, are based on the repeated use of an equilateral triangle as given below,
1. Divide each side of the equilateral triangle into three segments.
2. Draw an equilateral triangle (of smaller size) with sides the length of the line segments above, with the

base placed at the middle line segment, pointing outwards.
3. Remove the line segment forming the base of the newly placed triangle.

PROGRAM DEVELOPMENT PROBLEMS

D1. Coin Change Program (Revisited)
Consider the Coin Change program in section 3.4.6 (for children learning to count change) in which the user had to enter a combination of coins that added up to a specified amount between 1 and 99 cents. Develop and test a program based on the use of recursion that “turns the tables” in which the user enters an amount between 1 and 99 cents, and the program must determine a set of coins

that adds up to that amount using the least number of coins.

D2. Palindrome Checker (Revisited)
In Chapter 7 (section 7.3.7) a Palindrome Checker program was given based on the use of a stack that was also developed. Develop and test a Palindrome Checker program based on the use of recursion, instead of a stack.

D3. Phone Number Spelling Program (Revisited)
In Chapter 9 (section 9.1.2) a Phone Number Spelling program was given based on the use of deeply nested for loops. Develop and test a Phone Number Spelling program based on the use of recursion instead.

D4. Creative Designs
Using your imagination, develop and test a program using recursion that generates a fractal of your own design. As with the example fractals in the chapter, start with a basic geometric shape, and define the next level of the fractal pattern containing two or more smaller versions of the original-size shape.

D5. Develop a program that uses recursion to solve the Man, Cabbage, Goat, Wolf problem from Chapter 1.

Computing and Its CHAPTER 12 Developments

In this final chapter, we look at the many developments and innovations that have led to present-day computing. This includes advancements in computer hardware, computer software, and computer networks, as well as theoretical developments that underlie our understanding and effective utilization of computing.

OBJECTIVES
After reading this chapter you will:
♦ Be knowledgeable about many of the developments in computing ♦ Become familiar with some of the most notable individuals in the field ♦ Be able to describe the four generations of computer technology

CHAPTER CONTENTS
Contributions to the Modern Computer
12.1 The Concept of a Programmable Computer

12.2 Developments Leading to Electronic Computing

First-Generation Computers (1940s to mid-1950s)
12.3 The Early Groundbreakers
12.4 The First Commercially Available Computers Second-Generation Computers (mid-1950s to mid-1960s)
12.5 Transistorized Computers
12.6 The Development of High-Level Programming Languages Third-Generation Computers (mid-1960s to early 1970s)
12.7 The Development of the Integrated Circuit (1958)
12.8 Mainframes, Minicomputers, and Supercomputers

Fourth-Generation Computers (early 1970s to the Present)
12.9 The Rise of the Microprocessor
12.10 The Dawn of Personal Computing
The Development of Computer Networks
12.11 The Development of Wide Area Networks
12.12 The Development of Local Area Networks (LANs)
12.13 The Development of the Internet and World Wide Web

CONTRIBUTIONS TO THE MODERN COMPUTER
12.1 The Concept of a Programmable Computer
12.1.1 “Father of the Modern Computer”—Charles Babbage (1800s)

FIGURE 12-1 Charles Babbage

In London, England, during the 1800s, **Charles Babbage** (Figure 12-1) worked on a couple of different designs for a calculating machine. The first, called the "Difference Engine," was designed to perform only certain calculations. It was to be powered by steam with a built-in "printer" that would punch out calculation tables on metal plates, important for British navigation. A partial prototype was finished in 1822. Because of technical, financial, and other problems however, his continued attempts over the next nearly twenty years to finish its construction ended in failure. If completed, it would have measured ten-feet by tenfeet by five-feet, and would have weighted two tons.

Another machine that Babbage envisioned, however, would lead to his historical imminence—the **Analytical Engine** (Figure 12-2). He completed the first workable prototype in 1837,

working on its completion until his death in 1871. A major conceptual breakthrough of the Analytical Engine was that the calculations it performed were based on a set of instructions fed to it. Therefore, it could be "programmed" to solve *any* mathematical problem, not just certain problems as with the Difference Engine. As designed, it would have been fifteen feet tall and twenty-five feet long; about the size of a small locomotive train.

The Analytical Engine was never completed due to the limited technology of the time. Its design, however, is considered one of the greatest intellectual achievements of the nineteenth century. It contained the two fundamental components of current-day computers: a *mill* (a kind of central processing unit for executing instructions, fed as punched cards), and a *store* (a kind of memory or storage area). Because of Babbage's foresight, he has been given the de

served name "Father of the Modern Computer."

FIGURE 12-2 The Analytical Engine

Charles Babbage is considered the “Father of the Modern Computer” for his work on the Analytical Engine in the 1800s.

12.1.2 “The First Computer Programmer”—Ada Lovelace (1800s)

Augusta Ada Byron (Figure 12-3), Countess of Lovelace (**“Ada Lovelace”**) and daughter of the poet Lord Byron, was a talented mathematician, who first met Charles Babbage in 1833. She was intrigued with his work, and inspired Babbage to continue with his Analytical Engine. She helped write widely published scientific articles on the machine and Babbage’s ideas. She showed her insightfulness by predicting that such machines were capable of processing not only numbers, but any encoded information. For example, she gave predictions that someday these machines would be able to produce graphics, compose music, and have scientific use (beyond mere scientific calculation). Because of her foresight, she is credited with being the “First Computer Programmer.” The **Ada programming language** is named in her honor.

FIGURE 12-3 Ada Lovelace

Ada Lovelace is considered the “First Computer Programmer” for her work, insight and writings on the Analytical Engine.

12.2 Developments Leading to Electronic Computing 12.2.1 The Development of Boolean Algebra (mid-1800s)

We have already mentioned George Boole (Figure 12-4) in Chapter 3 in our discussions of Boolean algebra. His development of what we now call Boolean algebra, and the corresponding Boolean operators AND, OR, and NOT was conceived as a means of mathematically proving the answers to any true/false question. Later work showed that there are always some questions that cannot be answered mathematically. However, Boolean algebra became greatly applicable one hundred years later. It provides the logical foundation for the design of digital logic circuits of modern electronic computers.

FIGURE 12-4 George Boole

The development by **George Boole**, of what is now called Boolean algebra,

provided the logical foundation for the development of digital circuits.

12.2.2 The Development of the Vacuum Tube (1883)

FIGURE 12-5
Thomas Edison

In 1883, **Thomas Edison** (Figure 12-5) invented the **vacuum tube** (Figure 12-6), an electronic device containing electrodes that electrons can travel between. The vacuum tube became the building block for the entire electronics industry at the time. A version of the device can be used as an electronic switch, which became a critical component in the development of the first modern electronic computers.

FIGURE 12-6 Vacuum Tubes
Thomas Edison invented the vacuum tube, which became the first form of electronic switch in electronic computers.

12.2.3 The Development of Digital Electronic Logic Gates (1903)

Serbian-born **Nikola Tesla** (Figure 12-7), who once worked for Thomas Edison,

patented in 1903 the **electronic logic gate**. Such gates, composed of a small number of vacuum tubes as electronic switches, could electronically execute the logical operators of Boolean algebra—AND, OR, and NOT. Two different electrical signals representing either true or false were input to the gate, with an output electrical signal representing the true/false result, as shown in Figure 12-8. Logic gates would become fundamental to all electronic digital computers to come, the idea to be rediscovered decades later.

FIGURE 12-7 Nikola Tesla

FIGURE 12-8 AND Logic Gate

Nikola Tesla patented the electronic logic gate, which would become a fundamental component of all electronic computers.

12.2.4 The Development of Memory Electronic Circuits (1919)

American physicists **W. H. Eccles** (Figure 12-9) and **R. W. Jordan** in 1919 invented the **flip-flop** electronic switching circuit. These devices, based on use of vacuum tubes, were able to “flip and flop” from one stable state to another. Such a device is always in one of two states, thus provides a form of electronic storage, or “memory.” Flip-flop electronic components, like Tesla’s electronic logic gates, would become key components in the development of future digital electronic computers.

FIGURE 12-9
W. H. Eccles
W. H. Eccles and **R. W. Jordan** invented the flip-flop electronic switching circuit to become a fundamental component of all electronic computers.

12.2.5 The Development of Electronic Digital Logic Circuits (1937)

In 1937, **George Stibitz** (Figure 12-10) of Bell Laboratories, working on his kitchen table one night created the first digital electronic logic circuit. A digital electronic circuit is a circuit built out of electronic logic gates that can execute arithmetic or logical operations. Because such circuits are built out of combinations of logic gates, they are referred to as **combinatorial circuits**. Boolean algebra provides a mathematical basis for denoting, analyzing, and understanding such circuits.

The digital logic circuit that Stibitz built was for adding two binary digits, called a (full) **binary adder**. Such an adder is a fundamental component of all modern computers. Stibitz's work demonstrated the feasibility of building electronic logical and arithmetic circuits from the basic building blocks of electronic logic gates that Tesla earlier developed. This development, along with the flip-flop electronic circuit of Eccles and Jordan for the electronic storage of information, completed the technology needed for the design and construction of a fully electronic computer.

FIGURE 12-10 George Stibitz

George Stibitz created the first digital electronic logic circuit, constructed out of a combination of electronic logic gates.

12.2.6 “The Father of Information Theory”—Claude Shannon (1948)

FIGURE 12-11 Claude Shannon

While many of his colleagues were working on the hardware design of modern computers, **Claude Shannon** (Figure 12-11) at Bell Laboratories, worked on a theory of information to be processed by such machines. In 1937, at the age of 21, Shannon wrote a master’s thesis that showed how Boolean algebra can be simulated electronically by simple electronic switches to perform both logical and numerical calculations—the fundamental concept of digital computing

today. Some considered Shannon's result possibly the most important master's thesis of the twentieth century.

Later, in 1948, Shannon published a paper that laid the foundation for what would become the field of information theory. Shannon's *Fundamental Theorem of Information Science* states that *all* information can be represented by use of only two symbols, "0" and "1," which he called **bits** (b inary dig its). It has been said that the digital revolution began with Shannon's work.

It was therefore clear that electronic switches could both be used for the design of digital logic circuits (e.g., a circuit for adding two numbers), and for the storage of information of any kind, not only numerical values (as Ada Lovelace foresaw). Given Shannon's demonstration of the sufficiency of binary encoding for encoding all information, the stage was set for the development of electronic binary digital computing as we know it today. For his work, Shannon is known as the "Father of Information Theory."

Such binary coding of information was already in practical use for the transmittance of messages as audible electronic signals over wire. Earlier, in 1837, **Samuel Morse** (Figure 12-12) patented his design of a telegraph, which could electronically send messages (text) in the form of two signals, dots (e.g. short beep) and dashes (long beep), called **Morse Code**. Since only two sig

nals were used, this represented a form of binary encoding.

FIGURE 12-12 Samuel Morse

Claude Shannon developed the *Fundamental Theorem of Information Science*, stating that *all* information can be represented by only two symbols, "0" and "1,"

which he called bits (binary digits).

FIRST-GENERATION COMPUTERS (1940s–mid-1950s)

12.3 The Early Groundbreakers

12.3.1 The Z3—The First Programmable Computer (1941)

Konrad Zuse (Figure 12-13) built a number of binary computing devices starting in 1935, at a time when other computers were being built based on the decimal system. His initial machines were the **Z1**, the first mechanical binary digital computing device, and the **Z2**, the first fully functioning electromechanical computing device, completed in 1939 as World War II was approaching. Each of these devices had limited programmability, however.

While his design of binary computers with the Z1 and Z2 was significant, his most notable accomplishment was the development of the **Z3**, the first working electromechanical *programmable* computer, completed in 1941. The Z3, like the Z2, was an electromechanical device. After serving in the German army, he realized the computing potential of a fully electronic version of his computing devices. He submitted a proposal to the German army for funding for his next generation of machine, a fully electronic calculator, the **Z4**.

A fully electronic machine would have been thousands of times faster than his electromechanical versions. It would therefore have significant military use, such as for aircraft design and breaking coded messages. (We shall see that the Allies had crucial success here.) He estimated the project would take two years. His proposal was flatly turned down by the German Army Command since it was believed that Germany would soon win the war (which, of course, they lost a few years later in 1945). He went on to build the Z4 after the war, completing an electromechanical version in 1949 to become the world's first commercially available digital computer. He even wrote his own programming language for the Z4, called *Plankalkul,* with features of modern-day languages that was definitely ahead of its time.

FIGURE 12-13 Konrad Zuse

Konrad Zuse developed the first working programmable computing device, the **Z3**, an electromechanical binary digital computer, as well as the first commercially available electronic computer, the **Z4**.

12.3.2 The Mark I—First Computer Project in the United States (1937–1943)

Howard Aiken (Figure 12-14), a mathematics instructor at Harvard University, began a project in 1937 to build a general-purpose (i.e., programmable) electromechanical calculating machine. It was the first project for the development of a modern computer in the United States (developed without knowledge of the work of Konrad Zuse). Aiken, as opposed to most others in the early computing field, knew of Babbage's work, and therefore saw himself as following through with Babbage's unfulfilled dream. Now, however, Aiken had available current-day technology that Babbage did not.

The technology available at the time was electromechanical, making use of relay switches that mechanically switched on and off by control of electrical signals. The machine, therefore, was not a fully electronic computer, and thus limited in speed by the speed of the mechanical switching of the relay switches. IBM agreed to fund the project and supplied an engineering team to build the machine following Aiken's design. When the machine was finally finished in 1943, it was eight feet tall, fifty-one feet long, two feet thick, and weighed

FIGURE 12-14 Howard Aiken

five tons (Figure 12-15). It read instructions from (punched) paper tape, and data from punched cards.

Ironically, true to technological developments of today, the Mark I was obsolete the moment it was completed, since fully electronic computers were just on the horizon. It was, however, very newsworthy at the time, and a big press release was given to announce its completion. The Mark I turned out to be very reliable compared to other electronic computers. It was able to run 24 hours a day, seven days a week, and therefore was very productive. It was used over a period of 16 years. Its development has been called the “real dawn of the computer age” (in the United States).

FIGURE 12-15 Harvard Mark I Electromechanical Computer

The **Mark I** computer, designed by **Howard Aiken** at Harvard, was the first project for the development of a modern computer in the United States. Other, more advanced computers, however, were built before the completion of the Mark I.

12.3.3 The ABC—The First Fully Electronic Computing Device (1942)

FIGURE 12-16 John Atanasoff

J **ohn V. Atanasoff** (Figure 12-16), a physicist at Iowa State University, completed a design of a calculating device between 1939 and 1942 with graduate student **Clifford Berry**, called the **ABC** (Atanasoff-Berry Computer). It was the first to use vacuum tubes both for calculations (for the logic circuits) and for memory storage. Therefore, it was the first fully electronic computing device. However, since it was designed to solve only particular types of mathematical problems and could not be programmed, it was not a general- purpose computer.

The ABC was a binary computer, designed to store numerical values in base 2 (i.e., binary notation), and not base 10 as had all other machines previously built, with the exception of the computing devices developed

by Konrad Zuse in Germany. Because of the war, neither Atanasoff nor Zuse knew of the others' work. Therefore each independently hit upon the novel idea of a binary computer at essentially the same time.

A functioning prototype was finished in October 1939 that could do simple addition and subtraction in binary of eight digit decimal values. In 1942, the final version was completed and tested (Figure 12-17). It was the size of a desk, weighed 700 pounds, had over 300 vacuum tubes, and contained a mile of wire.

FIGURE 12-17 The ABC Was the First Fully Electronic Computing Device

After testing of the machine was completed, Atanasoff worked on assignments for the war effort, and no further development or use of the machine followed. Given that the ABC was not a programmable device, it lacked a key feature of other machines at the time, but is credited as the first fully electronic (binary) computing device.

John V. Atanasoff and **Clifford Berry** developed the **ABC** (Atanasoff-Berry Computer), a fully electronic binary computer. It was not, however, a programmable device.

12.3.4 Colossus—A Special-Purpose Electronic Computer (1943)

FIGURE 12-18 The Enigma (During World War II)

At the start of World War II, in 1939, Great Britain had decided that it would call upon the greatest mathematical and scientific minds available to break the ciphers that Germany was using for top-secret communications. The Enigma (Figure 12-18), developed at the end of World War I, generated complex coding

of messages based on constantly changing alphabetic substitutions. It was believe by the Germans to be an unbreakable code.

A secret project was begun in Great Britain to build a machine capable of breaking the Enigma code. The electromechanical machine was called **Bombe** (Figure 12-19).

This allowed, in particular, the breaking of German naval codes, and resulted in the significant reduction of shipping losses across the Atlantic to the United States, essential for the allied support. Later, however, a second generation of encryption was developed by the Germans that was significantly more complex, using a new encryption called Lorenz code. A second new, fully electronic machine was designed and built for breaking this new code.

The machine was called the **Colossus**, a fully electronic binary machine, completed at the end of 1943. A number of machines of this design were built and used (Figure 12-20). A second version, the Colossus II, was finished on June 1, 1944. A few days after its completion, the Allies were able to intercept and decode a message from the German High Command. The message indicated that the Germans had fallen for misinformation put out by the Allies that their landing point was an area called Calais, while in fact, their planned landing location was the beaches of Normandy. As a result, commander Dwight D. Eisenhower decided to go ahead with the Normandy invasion—the infamous D-Day—on June 6, 1944 (less than a week after the Colossus II was finished). It is believed by many that the ability to break German codes brought the turning point of the war in the Allies favor.

FIGURE 12-19 Bombe

FIGURE 12-20 The Colossus
The **Colossus**, built in Britain to break German codes during World War II, is believed by many to have turned the war in the Allies, favor.

12.3.5 ENIAC—The First Fully Electronic Programmable Computer The U.S. Army's Need for a Fully Electronic Computer (1943)

When the United States entered World War II in 1941, there was a sudden increased need of people who calculated ballistic firing tables, so-called "human computers" (Figure 12-21). In order to hit their target, gunners needed to aim their weapon at the proper angle and direction based on a number of factors including the distance of the target, the wind speed, wind direction, and air temperature. Gunners depended on such firing tables to find the correct angle of fire for all the current conditions for all the different ballistic missiles.

At the U.S. Army's Ballistic Research Laboratory at Aberdeen Proving Ground in Maryland, hundreds of human computers (mostly female mathematicians) were employed to calculate such tables. Two to four thousand possible trajectories had to be calculated for each pair of projectile and gun. Each human computer, using an electromechanical desk calculator, took almost three days to compute a single trajectory! By 1943, the Army was not able to keep up with the calculation of all the firing tables needed, and was desperately in need of a solution. Thus, the Army was willing to fund the construction of a fully electronic computer capable of solving this problem.

FIGURE 12-21 “Computers” During World War II

In 1943, the U.S. army was in desperate need of a means of computing the large number of firing tables needed for various ballistic missiles.

The Development of the ENIAC at the University of Pennsylvania (1945)

In August 1942, an assistant professor at The Moore School of Electrical Engineering at the University of Pennsylvania, **John W. Mauchly** (Figure 12-22) had written a paper on “The Use of High Speed Vacuum Tube Devices for Calculating.” While his idea was realized by others, he understood the kinds of computing speeds that such an electronic switching device could produce over electromechanical machines—tens of thousands of operations per second. He, along with a young electronics engineer named **Presper Eckert**, was awarded a contract by the U.S. Army in June 1943 to build their electronic calculating machine, with an agreement to build a duplicate of the machine at Aberdeen Proving Grounds. (The size of computers then did not allow them to be moved around— they were permanently located!)

FIGURE 12-22 John Mauchly and Presper Eckert

The **ENIAC** (for “Electronic Numerical Integrator and Computer”), shown in Figure 12-23, was finished in November 1945 (too late for its originally intended purpose) about three months after the end of the war.

It was eight feet high, eighty feet long, and weighed 30 tons. It used decimal notation, and thus was not a binary computer. When finished, however, it was roughly a thousand times faster than any other computer at the time. Whereas existing specialized calculators took fifteen to thirty minutes to compute the trajectory of a ballistic missile, the ENIAC took only twenty seconds, capable of 5,000 ten-digit additions per second.

As a demonstration of ENIAC’s general-purpose functionality, the ENIAC was reprogrammed to do crucial calculations on the development of the first hydrogen bomb (an advanced weapon of the atomic bomb was used in World War II). Programming on the ENIAC, however, did not mean sitting down and typing lines at a keyboard as is done today. It had to be physically reprogrammed by changing switches and reconnecting a patchwork of wires.

The data for the currently configured program was submitted on punch cards. Its biggest problem was the unreliability of the almost 18,000 vacuum tubes it contained. In fact, the reason that base ten representation was chosen over binary was that the binary approach would have taken many more such vacuum tubes. Although ENIAC proved itself to be extremely fast at the time, the unreliability of the vacuum tube was to remain one of its shortcomings.

Because of intellectual property rights disputes in 1946 with the University of Pennsylvania, John Mauchly and Presper Eckert decided to leave the university to form the Eckert-Mauchly Computer Corporation. They believed in the potential of commercial computers, whereas others did not see the need for more than one or two computers in the whole country. They went on to design and build one the first commercial computer in the United States, the UNIVAC I (shown in Figure 12-22).

FIGURE 12-23 The ENIAC Computer
John Mauchly and **Presper Eckert** developed the ENIAC, the first fully-electronic programmable computer in the U.S.

12.3.6 EDVAC/ACE—The First Stored Program Computers (1950)

Even before Eckert and Mauchly had completed the design of the ENIAC, they conceived of the idea of a stored-program computer. In the ENIAC, only the data was stored in memory (via punched cards). To execute diffferent programs, the ENIAC had to be rewired, and later was reprogrammed through a series of switches. In a stored-program computer, the program need only be entered once and stored in memory. This concept was an important next step since high-speed computers like the ENIAC were slowed by the need to be reprogrammed this way.

In 1944, Eckert and Mauchly received another contract from the army's Ballistic Research Laboratory, this time for the development of a stored-program computer to be called EDVAC. A distinguished mathematician from Princeton named **John von Neumann** (Figure 12-24) joined the project. Even though Eckert and Mauchly

FIGURE 12-24 John von Neumann

had earlier conceived of the idea of a stored- program computer, von Neumann developed a method of how it could work, and thus is generally cited as the originator of the idea. The *von Neumann machine* remains the predominant means of stored program execution today.

John von Neumann is credited with the concept of a stored-program computer. The first such computer built was called **EDVAC**. Its method of executing stored programs is utilized in essentially all computers today, called a **von Neumann machine**.

The British Stored-Program Computer—ACE (1950)

British scientists visited the Moore School of Electrical Engineering at the University of Pennsylvania soon after the end of World War II. On return to Britain, they worked on their own stored program computer named **ACE** (for "Automatic Computing Engine"). The lead designer of the project was one of the most influential and significant individuals in the history of computing, **Alan Turing**, who also played a key role in breaking German codes of World War II (Figure 12-25). A scaled-down version of the storedprogram machine was completed in 1950 (before the completion of the EDVAC).

FIGURE 12-25 Alan Turing

After the development of the EDVAC computer, Great Britain began worked on their own storedprogram computer named **ACE** (for “Automatic Computing Engine”). The lead designer of the project was one of the most influential and significant individuals in the history of computing, **Alan Turing**.

12.3.7 Whirlwind—The First Real-Time Computer (1951)

In December 1944, the U.S. Navy asked MIT to do a feasibility study of the development of a specialpurpose flight trainer to train pilots. This required the machine to be capable of real-time processing— that is, to compute fast enough to instantaneously respond to the actions of training pilots. A young gifted engineering graduate student, **Jay W. Forrester**, was offered the project. He changed the goal of the project from the development of a special-purpose device, to the design and development of a general electronic digital stored-program computer that could operate in real time.

The computer that Forrester developed was called **Whirlwind** (Figure 12-26). Although it operated in real time, it had one serious problem. The method of memory storage, consisting of thirty-two cathode ray tubes that rarely lasted more than a month, was unreliable, often putting the machine out of service.

FIGURE 12-26 The Whirlwind Computer

Forrester revolutionized the technology of memory storage by using magnetically charged, doughnut-shaped ceramic ferrite “cores” that could be electronically magnetized in a clockwise or counterclockwise direction (to represent “1” and “0”), shown in Figure 12-27. The cores were configured in a grid form, threaded with cross sections of wires so that any given core could be read or written by selection of the proper “vertical and horizontal” wires. The memory board in the figure is 6.5 inches square, and contained a storage capacity of 1024 bits. The term *core* is still used today—mainly in the term *core dump* referring to the raw display, or “dump” of the contents of main memory—even though computers no longer use magnetic core memory.

Besides the speed of this form of memory, another important feature was that each bit (or core) of memory could be read and written in the same amount of time. This was called *random access memory* (RAM), an essential characteristic of computer memory today. Core memory technology remained the main memory technology for almost thirty years.

FIGURE 12-27 One Plane of Magnetic Core

Memory
The Whirlwind, developed by **Jay W. Forrester**, was the first real-time computer, also introducing the use of core (random access) memory.

12.4 The First Commercially Available Computers

A new stage of computer history began when companies started manufacturing commercial computers. This was the first time that computers were not thought of as unique, one-of-a-kind machines, but as particular *model* computers. A new industry was beginning that many people did not see coming. This is reflected in a quote by Thomas Watson, president of IBM in 1943: “I think there is a market for maybe five computers in the world.” Two of the more notable commercially available computers first appeared at essentially the same time—the UNIVAC (mentioned earlier) in the United States, and the LEO, in Great Britain (discussed in the following).

12.4.1 The Struggles of the Eckert-Mauchly Computer Corporation (1950)

Eckert and Mauchly needed financial backing to produce their computer, the **UNIVAC I**. The Census Bureau was very interested in the development of this machine, and in 1946 gave the company their first contract to build a UNIVAC. However, the amount agreed on was not enough to cover all the development cost. By 1948, Eckert and Mauchly had five UNIVAC contracts but no completed machine, selling out to the Remington Rand Corporation in early 1950. Finally, in March 1951, the first UNIVAC (Figure 12-28) was built and delivered to the U.S. Census Bureau. Eventually, forty-six UNIVAC I computers were built and sold. The UNIVAC became very publicly known, appearing in cartoons and movies of the time. One notable event was when it was used on live TV to predict the U.S. presidential election

FIGURE 12-28 The UNIVAC I

between Dwight D. Eisenhower and Adlai Stevenson in the 1952 election (Figure 12-29).

Most pollsters had predicted a close election race. However, UNIVAC, using data from past elections, predicted a landslide win for Eisenhower. Given the unbelievability of the results, they did not announce this prediction to the TV viewers. As it turned out, Eisenhower won by one of the biggest landslides in history. Afterwards, later that evening, news commentator Walter Cronkite had to admit that UNIVAC had earlier made the right predication— “UNIVAC was right, we were wrong!”

FIGURE 12-29 UNIVAC Predicting the 1952 Presidential Election

The UNIVAC I was the first commercially available computer in the U.S.

12.4.2 The LEO Computer of the J. Lyons and Company (1951)

FIGURE 12-30 The J. Lyons and Company

The J. Lyons and Company of London, England, was one of the largest catering and food manufacturing companies in the world (Figure 12-30). After a trip to the United States in 1947 by two managers of the company, they realized that electronic computers would hold the key to improved efficiency of clerical procedures. Given that there was an ongoing computer project at Cambridge University, they offered some funding to the university for the project in return for advice on how to build their own computer. A recent Ph.D. graduate of Cambridge who had worked on the computer project there joined the Lyons company, and a team of technical employees was hired.

Work started on the construction of the computer, named **LEO** (for “Lyons Electronic Office”), started in January 1949 (Figure 12-31). The computer became operational in September 1951, just six months after the construction of the first UNIVAC computer in the United States. However, it took a couple more years before the machine was reliable enough

to market. It became, however, the “world’s first business computer.”

Soon, many companies heard of the machine, and it became a big success. At first, Lyons began “renting out” time on their machine. Eventually, Lyons started building machines for sale to others, and thus became, in addition to a food service company, a computer manufacturer. Various models of the LEO

computer sold moderately well until the 1960s, when American-built computers began dominating the UK computer market. Great Britain was soon after that out of the computer manufacturing business.

FIGURE 12-31 The LEO ("Lyons Electronic Office") I Computer

The **LEO** ("Lyons Electronic Office") was the first commercially available computer in the United Kingdom, with its production beginning in 1951.

SECOND-GENERATION COMPUTERS (mid-1950s to mid-1960s)

12.5 Transistorized Computers

12.5.1 The Development of the Transistor (1947)

The **transistor**, developed by **William B. Shockley**, **John Bardeen**, and **Walter H. Brattain** at Bell Telephone Laboratories in December 1947, is a solid-state, semiconductor device that enables the switching of electrical circuits "on" (e.g., "1") and "off ("0"). Thus in combination, transistors can be used to create logic circuits (such as the addition of two numbers). A transistor is referred to as a **solid-state device** because it is composed of solid material, in contrast to the previous switching technology of the vacuum tube. Vacuum tubes are similar to light bulbs in size, generate significant heat, and eventually "burn out," needing replacement. Solid-state devices do not burn out, do not generate significant heat, can be made arbitrarily small, and do not draw much power. A comparison of these two devices is shown in Figure 12-32.

Transistors are referred to as **semiconductors** because their electrical conductivity lies between that of insulators (like rubber) and conductors (like copper). The degree of conductivity can be ectronically altered, which is what makes them a suitable electronic switching device, as shown in Figure 12-33.

FIGURE 12-32 The Vacuum Tube vs. the Transistor

ECEC

current flows through positive current applied[B]
to base lead

negative current B barrier prohibits applied to base lead[current flow]

Transistor Switched On
(E – emitter C – collector B – base lead) FIGURE 12-33 Transistor Switching Devices

Transistor Switched Off
(E – emitter C – collector B – base lead)

Transistors operate through three connections, referred to as the *emitter*, the *collector*, and the *base lead*. The flow of current from emitter to collector is strictly determined by whether there is currently a positive voltage or negative voltage applied to the base lead. When a positive voltage is applied to the base lead, the material between the emitter and the collector becomes conductive, and electrons are free to flow, thus turning the connection “on.” Alternatively, when a negative voltage is applied to the base lead, material between the emitter and collector becomes nonconductive, preventing electrons from freely flowing, thus turning the connection “off.” These simple switching devices can be used in arbitrary combinations in order to produce any given digital logic combinatorial circuit desired. William B. Shockley, John Bardeen, and Walter H. Brattain (Figure 12-34) received the Nobel Prize in Physics in 1956 for their discovery of the transistor effect.

FIGURE 12-34 Bardeen, Brattain, and Shockley (Nobel Prize in Physics 1956)

The transistor is a solid state electronic switching device developed by **William B**. **Shockley**, **John Bardeen**, and **Walter H. Brattain**.

12.5.2 The First Transistor Computer (1953)

The world's first transistorized computer was completed in 1953 at the University of Manchester in the United Kingdom (Figure 12-35). A computer using all transistors as electronic switching devices rather than the vacuum tube standard at the time had a number of advantages, as mentioned above transistors do not burn out like they took much less space, and did not generated significant heat as compared to vacuum tube technology. In 1962, a transistorized computer that was developed by this research group was the fastest

in the world.

FIGURE 12-35 The First Transistorized Computer

The first fully-transistorized computer was completed at the University of

Manchester in 1953.

12.6 The Development of High-Level Programming Languages

12.6.1 The Development of Assembly Language (early 1950s)

In the mid-1950s, all of the computers at the time were extremely tedious to program. They needed to be programmed in either *machine code* or *assembly language*. **Machine code** (or **machine language**) is a numerical code and is the "native" language of the machine. For example, the numerical code "1001" might represent an add operation. **Assembly language** is a symbolic notation using

what are called *mnemonics* in place of the numerical codes of machine language —"add" might be used in place of "1001".

Although assembly language is somewhat better than machine code, it is still a low-level language. *Low level* means that each instruction performs a very simple task. For example, to accomplish the assignment A 5B 1C takes a number of assembly language instructions to accomplish. Assembly language represents some improvement over numerical machine code notation. However, programming in assembly language is still very tedious. Thus, *high-level* programming languages were developed, discussed next.

Assembly language is a symbolic notation using what are called *mnemonics* in place of the numerical codes of machine language.

12.6.2 The First High-Level Programming Languages (mid-1950s)

The use of assembly language, although an improvement over machine code, was only of minor help. Therefore, better programming languages needed to be designed. One of the most influential individuals in the history of computer science was **Grace Murray Hopper** (Figure 12-36). In 1951, working at Remington Rand (maker of the UNIVAC computer), she conceived a new type of programming language that could help "automate" the task of programming. The language was called a **high-level language** since the programmer could write instructions in a more natural form. For example, a programmer could write a single instruction such as A 5 B 1 C, without needing to break it down into the more primitive set of machine/ assembly language instructions.

Since computers are designed to execute only machine code, in order to execute

programs written in a high-level programming language, they need to first be translated into machine code. Such a translator is called a **compiler**. Whereas each line of an assembly language program is translated to one line of machine code, and thus a “one-to-one” translation, a compiler is capable of a “one-to-many” translation. Thus, one line of a program written in a high-level language is translated into many machine code instructions that accomplish it. The idea of a compiler that Grace Murray Hopper put forth was one of the most important in the development of modern computers.

IBM came out with the first programming language for commercially available computers in 1957 called **FORTRAN** (“FORmula TRANslation”). FORTRAN was a language suited for scientific programming. Other languages of note that were developed around the same time were **COBOL** (“Common Business Oriented Language”) based largely on the work that Hopper did at Remington Rand, meant for business processing needs; **ALGOL** (“ALGOrithmic Language”), for general computing; **LISP** (“List Processing Language”), well suited for developing artificial intelligence programs; and **BASIC** (“Beginners All-Purpose Symbolic Instruction Code”), meant as an easy-tolearn language to make computer programming accessible to all college students, developed by **John Kemeny** at Dartmouth College in 1963. BASIC became the first programming language available on the earliest personal computers, and by the end of the 1980s, millions of school children had learned to use it.

FIGURE 12-36 Navy Admiral Grace Murray Hopper

Grace Murray Hopper, in 1951, conceived of a new type of programming

language that could help “automate” the task of programming, referred to as *high-level languages*.

12.6.3 The First “Program Bug” (1947)

American engineers have been calling small flaws in machines *bugs* for over a century. Thomas Edison talked about bugs in electrical circuits in the 1870s. When the first computers were built during the early 1940s, people working on them similarly referred to “bugs” in both the hardware of the machines and in the programs that ran them.

In 1947, engineers working on the Mark II computer at Harvard University found a moth stuck in one of the components, causing the machine to malfunction. They taped the insect in their logbook and labeled it “first actual case of bug being found (Figure 12-37).” It has become a standard part of the language of computer programmers. The log book, complete with the attached bug, is on display at the Smithsonian Institution in Washington, D.C.

FIGURE 12-37 The First “Computer Bug”

In 1947, engineers working on the Mark II computer at Harvard University found a moth stuck in one of the components, causing the machine to malfunction. They taped the insect in their logbook and labeled it “first actual case of bug being found.”

THIRD-GENERATION COMPUTERS (mid-1960s to early 1970s) 12.7 The Development of the Integrated Circuit (1958)

While transistors had the aforementioned advantages over vacuum tube

technology, there remained the problem of how to wire together the increasing number of transistors needed for the increasingly powerful computers being designed, known as the “tyranny of numbers.” This was evident with the transistorized computers. The solution to this problem was the development of the integrated circuit (or semiconductor chip), in which such wiring of components became unnecessary.

Two different individuals, **Jack Kilby** at Texas Instruments and **Robert Noyce** at Fairchild Semiconductor, were both working hard on a solution to the problem. The elegant, practical solution that they eventually found was to replace the method of *wiring* together components on digital circuit boards with a method of *printing* the “wiring” onto a thin wafer of semiconductor material (as shown in Figure 12-38). Thus, transistors and circuits were “integrated” together in the manufacturing. This device, therefore, was called an **integrated circuit (IC)** (or semiconductor chip). This not only

FIGURE 12-38 Integrated (Printed) Circuit

FIGURE 12-39 Jack Kilby—Inventor of the Integrated Circuit

made circuits smaller, but much cheaper to mass-produce. By 1961, both companies were producing commercially available integrated circuits.

It is generally agreed that Jack Kilby was first to conceive of and develop the integrated circuit in 1958, and Robert Noyce is credited with developing practical manufacturing principles that allowed for its commercialization. Jack Kilby went on to invent the portable calculator in 1967. Robert Noyce (with Gordon E. Moore, who was a cofounder of Fairchild) started in 1968 a new company named Intel (for “Integrated Electronics”).

In the year 2000, Jack Kilby (Figure 12-39) was awarded the Nobel Prize in Physics for the invention of the integrated circuit. The importance of the development of the integrated circuit cannot be overstated. It forms the basis of all modern computing today, and is said to have caused a “second industrial

revolution.” It stands historically as one of the most important inventions of mankind.

“What we didn’t realize then was that the integrated circuit would reduce the cost of electronic functions by a factor of a million to one, nothing had ever done that for anything before.”—Jack Kilby

An example packaged integrated circuit (“chip”) is shown in Figure 12-40.

FIGURE 12-40 A Modern Integrated Circuit
The integrated circuit, invented and developed by **Jack Kilby** and **Robert Noyce**, stands as one of the most important inventions of mankind.

12.7.1 The Catalyst for Integrated Circuit Advancements (1960s)

While integrated circuits had been commercially available since 1961, computer manufacturers did not jump on the new technology because it meant completely redesigning their current transistorbased machines. However, there was a

politically motivated, impossible-seeming challenge that would be put forth to Americans that would speed up the development of this technology. America and the Soviet Union were about to enter a “space race.”

The Beginning of the Space Race

On October 4, 1957, the Soviet Union (now Russia) launched the first artificial satellite, named “Sputnik” (Figure 12-41), sending shock waves throughout the United States. In April 1961, the Soviets had their first successful manned flight, by Cosmonaut Yuri Gagarin, with many successful manned flights following. The United States had no space program.

In response, in October 1958, **NASA** (National Aeronautics and Space Administration) was created, exactly one year after Sputnik was launched. The first U.S. manned space flight occurred on May 5, 1961. Alan B. Shepard Jr. took a 15-minute suborbital

FIGURE 12-41 Sputnik Satellite

spaceflight that made him the first American in space. Less than a year later, on February 20, 1962, John H. Glenn Jr. became the first American to orbit the Earth. These were the first steps on a race to the moon.

On October 4, 1957, the Soviet Union (now Russia) launched the first artificial satellite, named “Sputnik” sending shock waves through the United States.
President Kennedy’s Challenge of Going to the Moon (1961)

In 1961, **John F. Kennedy** was inaugurated as president. It was the height of the cold war with the Soviet Union. Although Allies in World War II, the United

States and Soviet Union came out of the war distrustful of each other. The biggest arms race in history had begun.

President Kennedy, concerned about the lead that the Soviets had in space, decided to raise the goal. In a speech to a special joint session of congress on May 25, 1961 (Figure 12-42), 20 days after Alan Shepard's flight, Kennedy proposed:

FIGURE 12-42 President Kennedy's challenge to go to the Moon

"I believe that this nation should commit itself to achieving the goal, before this decade is out, of landing a man on the moon and returning him safely to the earth. No single space project in this period will be more impressive to mankind, or more important for the long-range exploration of space; and none will be so difficult or expensive to accomplish. . . . in a very real sense, it will not be one man going to the moon . . . it will be an entire nation. For all of us must work to put him there."—President John F. Kennedy

Thus, President Kennedy had in clear terms put out a challenge to the Soviets, a "space race," to see who could reach the moon first. It was on July 22, 1969, that Neil Armstrong, Michael Collins, and Edwin Aldren safely reached and returned from the moon (Figure 12-43)—just over five months from the end of the decade, within the timeframe that President Kennedy had proposed. What technological devel opments had to be made to make this happen in a little over eight years time? Because of the limited space and constrained environment of space flight,

the integrated circuit was the technology needed.

FIGURE 12-43 The Apollo 11 Astronauts—Armstrong, Collins, and Aldren

President Kennedy had in clear terms put out a challenge to the Soviets, a “space race,” to see who could reach the moon first. Because of the limited space and constrained environment of space flight, the integrated circuit was the technology needed.

The Crucially Needed Advancements in Integrated Circuits (early 1960s)

FIGURE 12-44 The Apollo Guidance Computer (AGC)

Integrated circuits were obviously the way to go to achieve Kennedy’s goal, given their reliability and reduced weight and size over current technology. The problem was that, in 1961, integrated circuits were in their primitive stages of development, and extremely costly, at about $1,000 per chip.

The project for the development of what became called the **Apollo Guidance Computer** (AGC) (Figure 12-44) was given to MIT. The integrated circuits that were needed pushed electrical engineers to work on the development of better and more cheaply produced chips. By

1964, the cost of each chip had dropped to $25. When it was finished for the first Apollo missions using the Saturn V rocket (Figure 12-45) the AGC weight only 70 pounds, contained 4,000 integrated circuits, measuring only two feet by one foot by six inches! It was the only computer in existence completely designed using integrated circuits. The requirements of the AGC drove technological developments in the design and production of integrated circuits. By 1973, computers built completely out of integrated circuits were commonplace. By the end of the decade, integrated circuits were developed containing tens of thou

sands of transistors on a chip.

FIGURE 12-45 The Saturn V Rocket

The challenge by **President John F. Kennedy** of putting a man on the moon was a catalyst for the advancements made in integrated circuits technology.

12.7.2 The Development of the Microprocessor (1971)

While integrated circuits were in use in computer systems, each chip was a special-purpose component used together in “chip sets.” A different set of chip sets was needed for each particular computing device, and the cost of designing chips was expensive.

By 1969, integrated circuits were becoming more and more advanced in capability. An engineer working at Intel, named **Marcian E. (“Ted”) Hoff**, hit upon a brilliant idea. Instead of Intel manufacturing numerous *special-purpose* chips for specific devices (such as handheld calculators or digital alarm clocks), why not build a *general-purpose* logic chip that could be programmed to perform any logical task. In this way, a single chip can satisfy the needs of any device. This was an advantageous approach for Intel, since they could strictly focus on the production of such general logic chips. Those incorporating them

into their products would be responsible for writing the programs.

This general logic chip was equivalent to the central processing unit (CPU) in mainframe computers. In mainframes computers, however, the CPU was

FIGURE 12-46 Marcian “Ted” Hoff (center), Inventor of the Microprocessor with co-developers Federico Faggin and Stanley Mazor

composed of a number of different circuit boards. A general logic chip compressed all those circuits onto one integrated circuit. Since this logic chip contained all the functionality of a central processing unit (the “heart” of any computer system), it became known as a **microprocessor**.

The first microprocessor was produced and made available on the open market in November 1971. It was developed with co-developers Federico Faggin and Stanley Mazor (Figure 12-46), as well as Masatoshi Shima (not shown). It was a four-bit processor called the 4004, and the first commercially available integrated circuit that could be programmed for different tasks. It contained all the necessary components of a central processing unit squeezed onto a chip one-sixth of an inch long and one-eighth of an inch wide, containing 2,250 transistors. (The term “microprocessor” was used to emphasize a complete CPU on a chip. While most CPUs are microprocessors, they are now simply referred to as a “processor.”) The 4004 was used in the first scientific handheld calculators produced by Hewlett-Packard in 1972.

Marcian E. (“Ted”) Hoff invented the microprocessor—a complete central processing unit on a chip.

12.8 Mainframes, Minicomputers, and Supercomputers

As computing became more and more of a commercial item, and used by various types of companies and organizations, the computing needs likewise became varied. Some required and could afford large computer systems, while others needed something less. We will see next how this diversity of needs was satisfied though the development of a more diverse set of commercial computer systems.

12.8.1 The Establishment of the Mainframe Computer (1962)

FIGURE 12-47 IBM 7090 Mainframe at NASA

As commercial computing developed, large computers became known as **mainframe computers** (or simply **mainframes**). Such system could consist of multiple cabinets, storing the central processing unit (before the development of the microprocessor), memory, tape drives for storage, and so forth, taking up a whole room. An example of an early mainframe was the IBM 7090, shown at NASA in 1962 in Figure 12-47.

In 1964, IBM came out with a system called the IBM 360 (Figure 12-48). It is considered by some to be one of the most successful computers in history. Various models of the 360 were developed until 1978. Its design influenced many other computer systems that followed.

The IBM 360 was actually a series of models that were compatible enough so that companies could “trade up” for a larger system without the need for significant reprogramming. It was the first series of computers of

fering a large range of computing needs. Customers had

FIGURE 12-48 IBM 360

the option of purchasing systems with various processor speeds and a range of main memory size, among other options.

12.8.2 The Development of the Minicomputer (1963)

FIGURE 12-49 Gordon Bell—"Father of the Minicomputer"

Until the early 1960s, all computers were large and expensive mainframes, costing millions of dollars, only affordable by universities and large companies. An MIT graduate, named **Kenneth Olsen**, who had worked in the research labs there, came to a realization that smaller, less expensive computers had a place in the world, so-called **minicomputers**. He formed his own company in the early 1960s, called Digital Equipment Corporation (DEC). Chief Engineer **Gordon Bell** (Figure 12-49) at DEC designed a series of computers

that were small and inexpensive, transistor-based single-user systems—the PDP series.

In, 1965, DEC produced a model called the PDP-8 (Figure 12-50). This was the first widely successful minicomputer, and is considered as ushering in the minicomputer age (and eventually, personal computers). The PDP-8 was an 11-

bit computer. Following the PDP-8 was the 16-bit PDP-11, and the 32-bit Vax-11. This series of computers proved to be very successful and longlasting. The last VAX system was manufactured in 2005.

A new dimension was added to the classification of systems—by size and cost. Prior to their development, companies and other organizations that were not able to afford their own computer had to purchase time-sharing from an available mainframe over very slow phone line connections. Now a new alternative was available.

In 2010, Bell received an honorary Doctor of Science and Technology degree from Carnegie Mellon University. The university referred to him as the "Father of the Minicomputer."

FIGURE 12-50 PDP-8 Minicomputer

Gordon Bell, in the early 1960s, designed the first minicomputers, and thus is called the "**Father of the Minicomputer**."

12.8.3 The Development of the UNIX Operating System (1969)

In the early 1960s a joint project was formed between General Electric, MIT, and Bell Labs to develop a time-shared computer called MULTICS. The project, however, ended without being as successful as hoped.

Work on the MULTICS project, however, motivated those involved from Bell Labs to continue towards the goal of interactive computing. Thus, they began work on their own time-sharing operating

FIGURE 12-51 Kenneth Thompson and Dennis Ritchie

system in 1969, to be known as **UNIX** (see www.bell-labs. com/history/unix/). The two individuals that developed UNIX were **Kenneth Thompson** and **Dennis Ritchie** (Figure 12-51).

During the development of UNIX, a new programming language named “C” was developed by Dennis Ritchie, evolved from an earlier language named “B” that Ken Thompson had developed. Eventually, the UNIX operating system was rewritten in the C language, which along with UNIX, became widely used.

The developments of UNIX and C, and their adoption by industry and academia, is one of the biggest success stories in computer science. Thompson and Ritchie received numerous prestigious awards, including the A.M. Turing Award from the Association for Computing Machinery (ACM) in 1983, the U.S. National Medal of Technology from President Bill Clinton (Figure 12-52) in 1999, and the Japan Prize for Information and Technology (2011).

UNIX and C (and its object-oriented successor **C 11**) are in much use today, including a personal computer version of UNIX called **Linux**.

FIGURE 12-52 Thompson and Ritchie Receiving the National Medal of Technology

Kenneth Thompson and **Dennis Ritchie** developed the high successfully **UNIX** operating system. In addition, Dennis Ritchie developed the **C** programming language evolved from the B programming language developed by Thompson.

12.8.4 The Development of Graphical User Interfaces (early 1960s)

Another major advance in operating systems design was the move from **text-based user interfaces** (in which all commands were typed) to **graphical user interfaces** (GUIs). A GUI interface provides the familiar computer user interaction by use of a mouse. It might surprise you that the **computer mouse** was invented and first demonstrated back in the early 1960s, by a professor at Stanford University named **Doug Engelbart** (Figure 12-53). He is shown here holding a prototype of the mouse, constructed out of a wood shell and two metal wheels. Soon afterward he demonstrated the first word processor and the first hypertext, the kind of links that now exist in web pages. He has received numerous awards, including the Lemelson-MIT prize—the world's largest single prize for innovation—and the National Medal of Technology, the nation's highest technology award.

Doug Engelbart invented the computer mouse in the early 1960s.

FIGURE 12-53 Doug Engelbart— Inventor of the Computer Mouse

12.8.5 The Development of the Supercomputer (1972)

FIGURE 12-54 Seymour Cray— “The Father of the Supercomputer”

A supercomputer, as the name implies, is a computer system designed for maximum computational power. Whereas mainframe computers generally timeshare computations over a number of users, supercomputers focus their power on executing a few programs as quickly as possible. They are used in solving problems that require massive numbers of mathematical calculations, such as for weather forecasting.

In 1972, Seymour Cray (Figure 12-54), the “Father of Supercomputing,” started a company called Cray Research. The CRAY-1, released in 1976, became one of the most successful supercomputers in history, setting a new standard for high-speed computing (Figure 12-55).

What is classified as a supercomputer is relative to current technological capabilities. A typical personal computer today would have been considered a “supercomputer” in the not so distant past. There is a website that each

year lists the top 500 supercomputers worldwide (see www.top500.org). These supercomputers have demonstrated computational speeds measured in “teraflops,” i.e., trillions of floating-point operations per second. The fastest supercomputer as of this writing is the Cray-XT5 HE, dubbed the “jaguar,” with a speed of 1.75 petaflops (almost 2 quadrillion floating-point operations per second), with 224,000 individual processors. It is installed at the National Center

for Computational Sciences at Oak

Ridge National Laboratories in Tennessee.

FIGURE 12-55 The CRAY-1

Seymour Cray is called the "**Father of the Supercomputer**."

FOURTH-GENERATION COMPUTERS
(early 1970s to the Present)

12.9 The Rise of the Microprocessor

12.9.1 The First Commercially Available Microprocessor (1971)

The development of the personal computer (originally "microcomputer") has been one of the biggest technical revolutions in history. It didn't emerge from the research labs of major companies and universities, but rather from a group of hobbyists interested in building and owning their own computers.

Before the development of the microprocessor in the early 1970s, it was not possible for individuals to build their own computer system. A central processing unit (CPU) was too technically sophisticated for hobbyists to build. The first microprocessor, the 4-bit 4004 released by Intel in 1971, was only powerful enough for handheld calculators. But in 1974, Intel came out with the 8080 processor (Figure 12-56). The 8080 was an 8-bit processor with a clock speed of 2 MHz (thousands of times slower than a typical CPU in personal computers today) and contained about 6,000 transistors (compared to billions of transistors today). It was powerful enough, however, to build a computer system around. Thus, the age of the personal computer was about to begin. We look at the development of the personal computer next.

FIGURE 12-56 The Intel 8080 Microprocessor

The **first commercially available microprocessor** was the **Intel 4004**, a four-bit processor released in 1971.

12.9.2 The First Commercially Available Microcomputer Kit (1975)

The first commercially available microcomputer was a kit (also offered fully assembled) called the **Altair 8800** (Figure 12-57), first advertised in January 1975 on the cover of *Popular Electronics* magazine. It consisted of a box with levers and lights, used to input and display information in binary notation (i.e., light on for "1," light off for "0"). This was, especially by today's standards, a very crude and simple computer. However, given that no one until then could say that they had their own "personal computer," it was a

thrill for many hobbyists.

FIGURE 12-57 The Altair 8800—The First Microcomputer Kit

The **Altair** was the **first available microcomputer** (personal computer), appearing in 1975.

12.10 The Dawn of Personal Computing

12.10.1 The Beginnings of Microsoft (1975)

In 1975, the Homebrew Computer Club began in the San Francisco area. Among the soon to be infamous members were **Paul Allen** and **Bill Gates** (Figure 12-58), who created a version of the BASIC programming language for the new Altair system, and comprised the beginnings of Microsoft. Also present at the club was **Gary Kildall**, who wrote the first programming language and disk

operating system for microcomputers, among many other contributions.

In the mid-1980s, Microsoft produced the MS-DOS operating system for personal computers, which dominated the market.

MS-DOS was a text-based operating system.

FIGURE 12-58 Bill Gates and Paul Allen

In the 1990s, Microsoft came out with the GUI-based Windows operating system, which remains the most widely used operating system today.

In 1975, **Bill Gates** and **Paul Allen** started the Microsoft Corporation. **Gary Kildall** wrote the first programming language and disk operating system specifically for microcomputers.

12.10.2 The Apple II (1977)

FIGURE 12-59 Steve Jobs and Steve Wozniak

Two other members of the Homebrew Computer Club were **Steve Jobs** and **Steve Wozniak** (Figure 12-59). In 1977 they went on to create the most successful personal computer at the time, the **Apple II** (Figure 12-60) and form the Apple Computer company. One of the Apple II's main features was the ability to display color graphics, displaying output on a standard television

monitor. Various models of the Apple II were produced until 1993.

FIGURE 12-60 The Apple II
In 1977, **Steve Jobs** and **Steve Wozniak** created the most successful personal computer at the time, the **Apple II**.

12.10.3 IBM's Entry into the Microcomputer Market (1981)

IBM, which first had doubts about the future of microcomputers, saw the success of the Apple II computer and decided to develop their own microcomputer. In 1981, they came out with the **IBM-PC** (for "Personal Computer"), and before long, had sold 1 million machines (Figure 12-61). By the end of the 1980s, 65 million PCs were being used in offices, homes, and schools. Because of their size, reputation, and sales force, IBM dominated the personal computer market. Eventually, the generic name for microcomputers became "personal computer."

FIGURE 12-61 IBM-PC in 1981
In 1981, IBM came out with the IBM-PC ("personal computer"). By the end of the 1980s, 65 million PCs were in use.

12.10.4 Society Embraces the Personal Computer (1983)

In a very unconventional selection, *Time Magazine* selected the personal

computer for its January 3, 1983 “Man (Machine) of the Year” cover article. In the article, it said that by the millions, computers were finding their way into offices, schools, and homes. In addition, it discussed a survey showing that the average American believed that “in the near future,” computers would be as commonplace as TVs and refrigerators. The public image of what computers were had made a big change. The article stated (even in 1983) that the 30-ton ENIAC computer “could now fit on a chip the size of a pea.” (In fact, as part of the fiftieth anniversary of the ENIAC at the place of its birth, the University of Pennsylvania, a group of students did just that. They made an exact logical replication of the original ENIAC on a chip about 1/3” by 1/5” in size.)

In 1983, given the widespread use and impact of the personal computer, Time Magazine selects the **personal computer as the “Man” of the Year**.

12.10.5 The Development of Graphical User Interfaces (GUIs) An Early Machine Using a Mouse Driven Graphical User Interface (1975)

Because computers at the time were not powerful enough to support the computationally intensive graphical interactive method of computing (as demonstrated by Doug Engelbart), the idea did not initially catch on. However, in 1975, a computer was developed at Xerox Palo Alto Research Center (Xerox-PARC) that incorporated the use of a mouse device into a rather novel computer called the **Alto**. Its screen was the dimension of a normal sheet of paper, and had a graphical user interface. Although this was a notable advancement by those working at Xerox research center, it was never marketed by Xerox, leaving many of the developers feeling frustrated. Some years later, however, in 1979, Steve Jobs visited Xerox-PARC, and after seeing the Alto and its interactive mouse, immediately knew that the future of computing was about to change.

In 1975, a computer was developed at Xerox Palo Alto Research Center (Xerox-PARC) that incorporated the use of a mouse device into a rather novel computer called the **Alto**.
The First Commercially Successful Computer with GUI/Mouse (1984)

In January 1984, Apple came out with the first commercially successful computer with a graphical user interface, the **Macintosh** (Figure 12-62). In 1983, Apple had come out with another mouse/GUI system named Lisa aimed at the business market, its high selling price kept it from being commercially successful. The Macintosh, on the other hand, was priced for individuals, and

highly marketed. In one of the most famous commercials in the history of television during the 1984 Super Bowl game, Apple created an image of a new wave of computing, awaking large groups of robotlike, hypnotized individuals, meant to represent the awakening of the “blind” followers of IBM (which had not yet come out with a GUI-based machine).

FIGURE 12-62 Apple Macintosh in 1984
In January 1984, Apple came out with the first commercially successful computer with a graphical user interface, the **Macintosh**.

Mouse-Driven GUI Operating Systems Predominate (1995)

Microsoft began releasing its GUI interface operating systems, Windows, in 1985 with the release of Windows 1.0. However, its Windows operating system for the IBM-PC (and clones) did not begin to catch on until the release of Windows 3.0 in 1990, some waiting until the release of Window 3.1 in 1993. However, when the much promoted and completely redesigned Windows 95 was released in August 1995, within six weeks 7 million copies had been sold. The GUI-based computing paradigm of computing had become mainstream.

With the release of the **Microsoft Windows** operating system the GUI-based computing paradigm of computing had become mainstream.

12.10.6 The Development of the C **11** Programming Language

We take the opportunity here to mention one of the most widely used programming languages today, C 11 (pronounced “cee plus plus”). The other widely used programming language is Java (discussed in section 12.13.4).

C 11 was developed by Bjarne Stroustrup in the early 1980s at Bell Labs (Figure 12-63). It was designed essentially as an extension of the C programming language, also developed at Bell Labs over a decade earlier. The most significant feature added to C 11 is the ability to use classes, therefore facilitating object-oriented programming. However, as opposed to most object-oriented

programming languages, such as Java, the use of classes is not required. Therefore, C 11 program can be written strictly in the procedural style. In this way C 11 is a hybrid language. There is some playfulness in the name of the language. The operator symbol “ 11” that exists in C (and C 11)

FIGURE 12-63 Bjarne Stroustrup— Inventor of the C11 Programming Language

increments a value by 1. Since Stroustrup’s new language was “an increment” of C, the name C 11 was chosen.
Bjarne Stroustrup invented C11 at Bell Labs in the early 1980s as an extension of the C programming language. C11 has become one of the most widely used programming languages today.

THE DEVELOPMENT OF COMPUTER NETWORKS
12.11 The Development of Wide Area Networks

12.11.1 The Idea of Packet-Switched
Networks (early 1960s)

During the early 1960s, **Leonard Kleinrock** (Figure 12-64) at MIT published work on the notion of a **packet-switched network**, in which communications to be sent on a network are divided into equal-sized packets and transmitted individually. When all packets are received at the destination, they are then reassembled into the original complete communication. This is how all communication on the Internet is done. (The first experiment of having two computers—one in Massachusetts, the other in California—communicate over standard (non-packet-switched) telephone lines failed, proving Kleinrock’s claim of the need for a packetswitched approach correct.) Because of the distance

covered by these networks, they became known as **wide-area networks (WANs)**.

FIGURE 12-64 Leonard Kleinrock—Inventor of Internet Technology

Leonard Kleinrock invented the notion of a packet-switched network, which is how all communication on the Internet is performed.

12.11.2 The First Packet-Switched Network: ARPANET (1969)

What was needed next for wide-area computer networks was a standard "language" or communications protocol that all computers on a network could understand. By the end of 1968, a number of individuals working on packet-switched communications received funding from a military defense department, resulting in a network called **ARPANET**, becoming the first packet-switched network.

In September 1969, the first Arpanet node was installed at the University of California at Los Angeles (UCLA). Another node was installed at Stanford, and in October 1969, the first "host to host" message was sent directly from one computer to another—the beginnings of what we now know as the Internet. Over the years, more universities and companies were added to the network. In October 1972, the first public demonstration of **email** communication occurred. For the next ten years, email became the most used aspect of the new ARPANET.

ARPANET was the first packet-switched network, becoming operational in

1969. The first public demonstration of **email** was in 1972.

12.12 The Development of Local Area Networks (LANs)

12.12.1 The Need for Local Area Networks

The term “wide-area network” is meant to convey communicating computers over a wide area (e.g., globally). This concept first emerged during the era of mainframe computers (in the 1960s). However, computer networking had new emerging needs with the rise in the use of minicomputers during the 1970s, and personal computers during the 1980s.

Before the use of personal computers, for example, employees in a company could communicate and share files through remote terminal screens to the company’s mainframe computer. However, during the 1980s, when the personal computer began appearing on everyone’s desk, something was lost. Now each employee had all the computing power that they needed right at hand, but their machine was isolated from all others in the company, each in their own “computing island.”

Computer networking had new emerging needs with the rise in the use of minicomputers during the 1970s, and personal computers during the 1980s.

12.12.2 The Development of Ethernet (1980)

Robert Metcalf (Figure 12-65) was working as a researcher for the Xerox Corporation in the early 1970s when he was asked to work on a particular problem. Xerox was in the process of developing the world’s first laser printer, and wanted all the computers (hundreds of them) at the Xerox-PARC research facility where he worked to be connected to it. This problem called for a means of communication that was *fast* and allowed *many communications at once* from the hundreds of computers in the building.

His solution was the development of **local area networks (LANS)**. Local area networks are defined as a network of computers and computing devices (such as printers) within about a one-mile radius. As opposed to wide-area networks, in which the topology (i.e., the connections between computers) allows messages to be routed many different ways, Metcalf developed a network based on a simple topology. His topology consisted of a single, high-speed connection that all computing devices could share, called **Ethernet**. Ethernet was made

commercially available in 1980, and is now the most widely used local area network standard. Robert Metcalfe was awarded the National Medal of Technology from President George W. Bush for his work in 2003.

FIGURE 12-65 Robert Metcalf—Inventor of Local Area Networks

Robert Metcalfe invented the concept of a local area network, which led to the development of Ethernet, the most widely used local networking standard today.

12.13 The Development of the Internet and World Wide Web

12.13.1 The Realization of the Need for “Internetworking”

The development of ARPANET proved the viability of packet-switched computer networking. However, it had its own specific protocol (i.e., specific means of communication between computers). The future was viewed, however, as developing into a number of different kinds of packetswitched networks, each with their own unique protocols. The idea of “internetworking” was the ability of individual networks, normally unable to communicate with each other, to be able to communicate using an “internetworking protocol” (IP), resulting in one big computer network.

Internetworking is the ability of individual networks, normally unable to communicate, to communicate using an “internetworking protocol” (IP).

12.13.2 The Development of the TCP/IP Internetworking Protocol (1973)

A research program was initiated in 1973 to investigate ways of internetworking different packet-switched networks. The standard internetworking protocol that evolved was called TCP/IP. Since packets of a given communication travel independently in a packet-switched network, TCP/IP had the responsibility of ensuring that each packet arrived at its destination intact and reassembled into the complete original communication. The development of the TCP/IP protocol was developed by **Vinton G. Cerf** and **Robert E. Kahn** (Figure 12-66) referred to as the "Fathers of the Internet." In 2005 they each received the Medal of Freedom from President George W. Bush for their work.

On January 1 1983, TCP/IP became the standard protocol for the internetworking of networks. On that date, all computers currently connected to the Internet (using older protocols), smoothly switched to the new protocol simultaneously, and thus was the birth of *the* **Internet** as we know it today. The Internet is now comprised of a collection of over 50,000 independent networks, on all seven continents, and even planned for use in outer space.

FIGURE 12-66 Vinton G. Cerf and Robert E. Kahn—"The Fathers of the Internet"

Vinton G. Cerf and **Robert E. Kahn** developed a common network protocol called **TCP/IP** that allowed for the internetworking of computer networks. Thus, they are referred to as the "Fathers of the Internet."

12.13.3 The Development of the World Wide Web (1990)

The idea of having text that can be clicked on to lead one to more text has goes all the way back to a system developed in 1945, providing links between documents on microfiche. Doug Engelbart in the early 1960s demonstrated the notion of "mouse-clickable" text. Later, the term "hypertext" was coined. **Hypertext** documents are essentially "three-dimensional text," in that there are

words “behind” other words when clicked on.

FIGURE 12-67 Tim Berners-Lee—“Father of the World Wide Web”

12.13 The Development of the Internet and World Wide Web 523

Tim Berners-Lee (Figure 12-67), while working at CERN in 1980 (a European center for nuclear research outside of Geneva, Switzerland), developed a simple program with hypertext properties. He later revived his idea and in 1990 developed a program he named “WorldWideWeb,” the first web browser and hypertext editor. He is thus known as the “Father of the World Wide Web.”

A **web browser** is an application program to properly display hypertext (web) pages, performing the necessary actions of mouse events (e.g., mouse click on a link). In May 1991, the WideWorldWeb program was released for use at CERN for the creation, searching, and cross-referencing of research

papers in theoretical physics.
In 1993, a web browser was developed at the National Center for Supercomputing Applications (NCSA) at the University of Illinois named **Mosaic**, developed by **Marc Andreessen** (Figure 12-68). This was the first browser readily available to the general public, credited with popularizing the World Wide Web. Andreessen later went on to form Netscape Corporation, developing the Netscape family of browsers.

FIGURE 12-68 Marc Andreessen—Developed the First Readily Available Web Browser

Tim Berners-Lee was the inventor of the World Wide Web. **Marc Andreessen** developed Mosaic, the first web browser readily available to the general public. He went on to form Netscape Corporation.

12.13.4 The Development of the Java Programming Language (1995)

The development of the World Wide Web (and Internet browsers such as Netscape) was a very significant advance for the sharing of information. However, the information was static. One could interact with the browser by clicking on a link of one page of (static) information to retrieve another page. However, the web pages themselves did not have much capability of interacting with the user. For example, a web page offering a mortgage calculation would get the input from the user, then send it back to the server for calculations to occur there, with the results downloaded in another static page. Thus, there was not a way to download within a web page programs capability of performing significant computation on the user's computer (the "client").

As a matter of coincidence, a language perfectly suited for embedding programs in web pages emerged at the same time as the web. This was the **Java programming language**, was developed by James Gosling (Figure 12-69) at Sun Microsystems during the early 1990s.

FIGURE 12-69 James Gosling—Inventor of the Java Programming Language

A critical feature needed for the development of dynamic web pages was that the language used had to be executable (“understood”) by all the different kinds of computers on the Internet if the capability were to be available to all. The historical coincidence that occurred was that in 1990, Sun Microsystems began work on a new programming language written with the same capability of being able to be “complied once, run anywhere.” The need was related to software development for embedded systems, that is, systems containing an embedded processor, such as microwave ovens. With the new World Wide Web, it was realized the features of Java, developed for another purpose, were fundamentally suited for dynamic web pages. Java was expanded to include features especially suited for network applications, and was first released to the public in 1995, just a couple of years after the “web” became publically and freely available.

James Gosling invented the Java programming language at Sun Microsystems in the early 1990s. It has become one of the most widely used programming languages today.

Python 3 Programmers APPENDIX Reference

The author gratefully acknowledges the contributions of Leela Sedaghat to the style and content of this reference.

This reference provides the features of Python 3 that are most relevant for the text. Therefore, it is not intended to be an exhaustive resource. For complete coverage of the Python programming language, Standard Library and other Python-related information, we refer readers to the official web site:

http://www.python.org

A. Getting Started with Python 528
A1. About Python 528
A2. Downloading and Installing Python 528
A3. Program Development Using IDLE 529
A4. Common Python Programming Errors 539

B. Python Quick Reference 540
B1. Python Coding Style 541
B2. Python Naming Conventions 542
B3. Comment Statements in Python 543
B4. Literal Values in Python 544
B5. Arithmetic, Relational, and Boolean Operators in Python 545 B6. Built-in Types and Functions in Python 546
B7. Standard Input and Output in Python 549
B8. General Sequence Operations in Python 550
B9. String Operations in Python 551

B10. String Formatting in Python 552
B11. Lists in Python 553
B12. Dictionaries in Python 554
B13. Sets in Python 555

B14. if Statements in Python 556 B15. for Statements in Python 557 B16. while Statements in Python 558 B17. Functions in Python 559
B18. Classes in Python 560
B19. Objects in Python 561
B20. Exception Handling in Python 562 B21. Text Files in Python 563
B22. Modules in Python 564

C. Python Standard Library Modules 565 C1. The math Module 566
C2. The random Module 567
C3. The turtle Module 568
C4. The webbrowser Module 570

A. GETTING STARTED WITH PYTHON
A1. About Python

The Python programming language was created by Guido van Rossum (www.python.org/~guido) at the Centrum Wiskunde & Informatica (National Research Institute for Mathematics and Computer Science) in the Netherlands in the late 1980s. It is designed for *code readability*. It therefore has a clear and simple syntax. At the same time, Python also has a powerful set of programming features.

Python is free, open source software (http://www.python.org). The reference (standard) implementation of the Python programming language (called CPython) is managed by the non-profit Python Software Foundation. The language is bundled with a Python development environment called IDLE. The bundle is available for download at the official Python web site (see below). There are two, incompatible versions of Python currently supported: Python 2 (2.7.3) and Python 3 (3.2.3 at the time of this writing). Python 2.7.3 will be the last release version of Python 2. This text uses Python 3.

Python is growing in popularity. Many companies and organizations use Python including Google, Yahoo and YouTube. Python is also widely used in the scientific community, including the National Weather Service, Los Alamos National Laboratory, and NASA. Python also continues to gain popularity for use in introductory computer science courses.

A2. Downloading and Installing Python

To download and install the Python interpreter (and bundled IDLE program) go to http://www. python.org/download/ which displays the page shown in Figure A-1.

Download Python

The current production versions are Python 2.7.3 and Python 3.2.3.

Start with one of these versions for learning Python or if you want the most stability; they're both considered stable production releases.

If you don't know which version to use, start with Python 2.7; more existing third party software is compatible with Python 2 than Python 3 right now.

For the MD5 checksums and OpenPGP signatures, look at the detailed Python 2.7.3 page:

- Python 2.7.3 Windows Installer (Windows binary -- does not include source)
- Python 2.7.3 Windows X86-64 Installer (Windows AMD64 / Intel 64 / X86-64 binary [1] -- does not include source)
- Python 2.7.3 Mac OS X 64-bit/32-bit x86-64/i386 Installer (for Mac OS X 10.6 and 10.7 [2])
- Python 2.7.3 Mac OS X 32-bit i386/PPC Installer (for Mac OS X 10.3 through 10.6 [2])
- Python 2.7.3 compressed source tarball (for Linux, Unix or Mac OS X)
- Python 2.7.3 bzipped source tarball (for Linux, Unix or Mac OS X, more compressed)

Also look at the detailed Python 3.2.3 page:

- Python 3.2.3 Windows x86 MSI Installer (Windows binary -- does not include source)
- Python 3.2.3 Windows X86-64 MSI Installer (Windows AMD64 / Intel 64 / X86-64 binary [1] -- does not include source)
- Python 3.2.3 Mac OS X 64-bit/32-bit x86-64/i386 Installer (for Mac OS X 10.6 and 10.7 [2])
- Python 3.2.3 Mac OS X 32-bit i386/PPC Installer (for Mac OS X 10.3 through 10.6 [2])
- Python 3.2.3 compressed source tarball (for Linux, Unix or Mac OS X)
- Python 3.2.3 bzipped source tarball (for Linux, Unix or Mac OS X, more compressed)

FIGURE A-1 Official Download Site of Python

Select the appropriate Python 3 download for your system: Windows x86 MSI installer (for typical Windows machines), Windows x86-64 (for 64-bit Windows machines, not typical), Mac OS X 64-bit/32-bit x86-64/i386 (for newer Macs) and Mac OS X 32-bit i386/PPC (for older Macs). There are also versions for Linux and Unix, as shown. Follow the installation directions.

A3. Program Development Using IDLE

What is IDLE? IDLE is an *integrated development environment* (IDE) for developing Python programs. An IDE consists of three major components: an **editor** for creating and modifying programs, an **interpreter** or **compiler** for executing programs, and a **debugger** for “debugging” (fixing errors in) a program.

When you execute IDLE on your system, a window such as shown in Figure A-2 will be displayed.

FIGURE A-2 The Python Shell

Note that the version of Python is displayed on the lines right before the prompt (...).

Interacting with the Python Shell The Python Shell provides a means of directly interacting with the Python interpreter.

Thus, whatever Python code that is typed in will be immediately executed, as shown in Figure A-3.

FIGURE A-3 Interacting with the Python Shell

This is referred to in programming as *interactive mode*. All variables will remain defined until the shell is either closed or restarted. To restart the shell, select Restart Shell from the Shell dropdown list, as shown in Figure A-4.

FIGURE A-4 Restarting the Python Shell

After the shell is restarted, all previously-defined variables become undefined and a “fresh” instance of the shell is executed, as shown in Figure A-5.

FIGURE A-5 A New Instance of the Python Shell

Being able to immediately execute instructions in the Python Shell provides a simple means of verifying the behavior of instructions in Python. For example, if one forgot the specific range of numbers that built-in function range(start, end) produces for a given start and end value, the single for statement in Figure A-6 can be easily executed to determine that.

FIGURE A-6 Testing Small Program Segments in the Python Shell

We can also execute Python code in the shell that requires the use of modules. To utilize a particular module in the shell, the appropriate import statement is first entered.

FIGURE A-7 Importing a Module into the Python Shell

Finally, when working in the interactive Python Shell, there are helpful keyboard shortcuts that the user may need to use. Some of the most commonly used commands are listed in Figure A-8.

Windows Shortcut	Mac Shortcut	Action Performed
Ctrl+C	Ctrl+C	Terminates executing program
Alt+P	Ctrl+P	Retrieves previous command(s)
Alt+N	Ctrl+N	Retrieves next command(s)

FIGURE A-8 Python Shell Short c uts

Creating and Editing Programs in IDLE Instructions typed into the shell are executed and then “discarded.” For program development, however, we need to save instructions into a file. This is referred to as a **script** in Python. A program in Python is a script. To begin creating a script in IDLE, select New Window from the File dropdown menu, as shown in Figure A-9.

A Python program (or other text files) can be typed in this window. This program is saved by going to the File menu option and selecting Save (or by using shortcut key Ctrl 1S), depicted in Figure A-10.

When selecting the Save option on a currently unnamed file, a file window appears allowing a name to be entered for the new program (Figure A-11). If the program was already named, choosing Save would save the file under the current name, overwriting the currently saved file. Using Save As allows a file to be named under a different name, such as for creating a backup file.

FIGURE A-9 Opening a Script Edit or Window

FIGURE A-10 Entering and Saving a Python Program File

FIGURE A-11 The File Window and Naming and Saving a File

When saving a Python program, *remember to add the file extension .py to the filename*. Otherwise, the saved file will not appear in the file window when working with Python programs in IDLE. It also will not be considered a Python file by the operating system. In this case, Hello World Program.py will become a new file in the My Python Programs folder in addition to the existing file Some Program.py.

When creating and modifying programs, there are a number of editing features that IDLE provides. Many of these commands will likely be familiar to you. However, in addition to the familiar commands (such as “cut” and “paste”) there are editing features in IDLE that are specific to Python. These commands are listed in Figure A-12.

The backspace is for deleting characters *before* the cursor, and delete deletes characters *after*. Lines may be deleted, copied and pasted within a program. For navigating a program file, there is the ability to go to a specific line number, or search for lines containing a specific search string. Once a search has begun, the shortcut Ctrl 1G provides a convenient way to continue the search to each next line found.

Windows Shortcut	Mac Shortcut	Action Performed
Backspace Key	Delete	Erases character before cursor
Delete Key	-	Erases character after cursor
Ctrl+C	⌘+C	Copies highlighted set of lines to be pasted
Ctrl+X	⌘+X	Cuts highlighted set of lines to be pasted
Ctrl+V	⌘+V	Pastes most recently copied or cut highlighted lines
Alt+G	⌘+J	Prompts for line number to place cursor
Ctrl+F	⌘+F	Prompts for characters to search for
Ctrl+G	⌘+G	Continues current search
Ctrl+0	Ctrl+0	Shows the innermost matching parentheses
Ctrl+Z	⌘+Z	Undo all previous editing actions one-by-one
Ctrl+]	⌘+]	Indents selected lines
Ctrl+[	⌘+[	Unindents selected lines back to the left
Alt+3	Ctrl+3	Comments out selected lines
Alt+4	Ctrl+4	Removes ## notations from start of selected lines
Ctrl+S	⌘+S	Saves program as current file name

FIGURE A-12 Editing Commands in IDLE

The Show Matching command (Ctrl 10) identifies certain matching delimiters. The innermost set of matching parentheses, square brackets or curly braces are highlighted from the current position of the cursor. This is useful for identifying mismatched parentheses in expressions, as well as mismatched delimiters in data structures containing parentheses, square brackets and/or curly braces. The Undo command (Ctrl 1Z) will undo the most recent edit change in a file. This can be used to undo all the edit changes, even before the last time the file was saved, way back to the state of the file when it was first opened (or back to an empty file if being created).

One particularly useful pair of commands are the Indent / Dedent commands. These indent and “un-indent” a selected (highlighted) set of program lines. The number of spaces of indentation by default is four, which follows the Python convention for style. Therefore, it is recommended that you do not change this value. (It can be set under the Options / Configure IDLE menu selection.) Another useful set of commands is Comment Out and Uncomment. These commands add (and remove) a “double comment” symbol, ##, on a selected set of lines. This is useful for disabling certain portions of a program during program development and testing. A summary of some of the editing commands in IDLE is given in Figure A-12. The *shortcut keys given here are worth learning so that you can be more efficient in your programming*. Each command has a corresponding menu option.

We show all the menu options in the editor window of IDLE below. The **File menu**provides the ability to open, save and print files, shown in Figure A-13.

FIGURE A-13 The File Menu of the IDLE Edit or Window

The **Edit menu** provides options for the usual editing commands such as Undo/Do, Copy/Cut/ Paste, Find/Replace, shown in Figure A-14.

FIGURE A-14 The Edit Menu of the IDLE Edit or Window

The **Format menu** provides options for Indenting / Dedenting, and Commenting Out / Uncommenting a section of code, shown in Figure A-15.

FIGURE A-15 The Format Menu of the IDLE Edit or Window

The **Run menu** provides the Run Module, which executes the Python code currently in the editor window, shown in Figure A-16. *The F5 shortcut key is a convenient way to execute a program*.

FIGURE A-16 The Run Menu of the IDLE Edit or Window

The **Options menu** provides the option Configure IDLE for configuring various aspects of IDLE, including the fonts used, the color of keywords, the color of comment lines, etc., and the number of spaces used for each tab character (for indentation of program lines). Although there is a lot of control provided for the "look and feel" of IDLE, it is recommended to stick with the standard

FIGURE A-17 The Options Menu of the IDLE Edit or Window

configuration options. (Code Content is a feature intended to help aid in keeping track of the lines in the same program block while scrolling through a file. We do not see the benefits of this option for our purposes.)

The **Windows menu** provides options for controlling the height of the window. It also provides a list of all currently open Python windows (by file name), including the Python Shell. This provides an easy way to switch from one window to another. The menu options are shown in Figure A-18.

FIGURE A-18 The Windows Menu Options of the IDLE Edit or Window

Finally, the **Help menu** includes About IDLE, which includes the version number of IDLE installed; and IDLE Help, which gives a brief summary of the commands of the menu bar.

FIGURE A-19 The Help Menu Options of the IDLE Edit or Window

Using Python Docs from within IDLE A very useful feature of IDLE is the **Python Docs** option under the Help menu. This option links to the official documentation for the version of Python installed, shown in Figure A-20. The most relevant parts of the documentation for this text are marked with a (dark) checkmark. The Tutorial starts with an introduction to Python, and goes through all the features of the language, thus covering material beyond the scope of this text. It is a good starting point, however, for obtaining more information on a particular language feature. The Library Reference lists all the built-in functions, constants and types. In addition, it contains a categorized list of the Standard Library modules in Python. Finally, the Python Language Reference contains all information about the “core” of the language. This includes documentation on general syntax, expressions, statements, and compound statements (such as if and while statements). The Global Module Index, General Index and Python FAQs, indicated by lighter check marks, may also be of some help to the reader.

A4. Common Python Programming Errors 537

FIGURE A-20 Python Docs

A4. Common Python Programming Errors

We list the typical errors for novice programmers using Python. It would be wise to use the following as a checklist when developing and debugging your Python programs.

♦ Improper Indentation

All instructions in the same suite (block) must be indented the same amount. Each tab press will move the cursor the number of character spaces that is set under Options / Configure IDLE (four spaces by default, recommended).

♦ Forgetting About Truncated (Integer) vs. Real Division
Keep in mind the difference between the / operator (5/4 → 1.25) and the // operator (5/4 → 1). This is very easy to forget when using arithmetic expressions.

Confusing the Assignment Operator (5) with the Comparison Operator (55) ♦
This is a very common error for new programmers, but one that you should learn to avoid as early as possible.

♦ Forgetting to Convert String Values when Inputting Data
Remember that the input function always returns a string type. It is easy to forget when reading in numeric values to convert them to integer or float before using arithmetically.

♦ Forgetting to Use Colons Everywhere Needed
Don't forget to put colons where needed. Rather than trying to memorize where they are required, a simple rule can be followed. A colon is required after any keyword (such as if or while) in which the subsequent statements are indented.

♦ Forgetting about Zero-Based Indexing
Zero-based indexing, in which the first index value of an indexed entity starts at 0, can lead to “off-by-one” errors if the programmer is not careful.

♦ Confusing Mutable and Immutable Types
Remember that the value of a mutable type can be changed without the need for reassignment, for example, list1.append(40). Since tuples and strings are immutable types and thus their values cannot be changed, reassignment of the variable is needed in order to “change” it. Thus, the statement str1 1 'There! ' does not change the value of str1. To change its value, it must be reassigned, str1 5 str1 1' There!'

♦ Forgetting the Syntax for Tuples of One Element
Tuples are the only sequence type that requires a comma with tuples of only element, (1,) → (1). If the comma is left out, then the expression evaluates to that element, (1) → 1.

♦ Improperly Ended Program Lines
Forgetting to use the backslash (\) when continuing a program line to the next line.
B. PYTHON QUICK REFERENCE

The references pages contained here summarize aspects of the Python programming language most relevant to the textbook. Therefore, the functions/operators listed, and the available optional arguments for each is not meant to be comprehensive. For complete coverage, see The Python Language Reference of the official Python site at http://docs.python.org/reference/index.html .

B1. Python Coding Style 539

B1. Python Coding Style

Use four spaces for each indentation level.

Use blank lines, sparingly, to separate logical sections of code. Separate function definitions with two blank lines. (Same for class definitions)

Limit length of lines to 79 characters (i.e., do not wrap lines around screen).

Statements containing open parentheses, square brackets or curly braces can be continued on the next line (except when containing open single or open double quotes):

```
result = (num1 + num2 + num3 + num4 + num5 +
num6 + num7 + num8 + num9 + num10)
```

Statements without such delimiters can be continued by use of the line continuation character (\):

```
response = \
int(input('(1)continue processing, (2)quit program '))
```

This: i = i + 1

Not This: i=i+1

This: c = (a + b) * (a - b) **Not This:** c = (a+b) * (a-b) **This:** (a, b, c, d)

Not This: (a,b,c,d)

This:

def func1(real, imag=0.0): **Not This:**

def func1(real, imag = 0):

This:

def func1(real, imag=0.0): **Not This:**

def func1(real, imag = 0):

B2. Python Naming Conventions

All uppercase, underscores used when aids readability: RATIO ANNUAL_RATE

All lowercase, underscores used when aids readability: n, line_count

All lowercase using underscores when needed, or mixed case (“camel case”), in which first character is lowercase, and first letter of all other words is uppercase: calcAverage, calc_average
Mixed case, with first character a capital letter: VehicleClass
Short name, all lowercase, underscores used when aids readability:
math, conv_functions

and del from None True as elif global nonlocal try assert else if not while break except import or with class False in pass yield continue finally is raise
def for lambda return

B3. Comment Statements in Python 541
B3. Comment Statements in Python

Comment line The hash sign (#) is used to make single line comments (when at the start of a line), or an in-line comment (when within a line). All characters following the hash sign until the end of line are treated as a comment.

In-line comment, e.g.,
state_tax_rate = 0.08 # 2008 tax rate
Single line comment serving as program section heading, e.g., # sum all values greater than 0
for i in range(0, len(values)):
if values[i] > 0:
sum = sum + values[i]
As an alternative to use of a triple-quoted docstring, e.g.,

def function1(n):
#----------# This function returns the largest integer # less than or equal to n.
#----------

Never include an in-line comment that states what is obvious from the program line, e.g.,
n = n + 1 # increment n
But this is ok, n = n + 1 # adjust n to avoid off-by-one error
If a # appears in a line with a prior #, then the second# is taken as part of the comment. There is no provision for block comments in Python (i.e., commenting out a sequence of lines with one set of comment delimiters).

B4. Literal Values in Python

None

None is a special “place marker” in Python that can be used when there is no available value.

Numeric literals never contain commas… **Yes:** 1200 204231
No: 1,200 204,231

Integer literals never contain a decimal point… **Yes:** 1200
No: 1200. 1200.0

There is no limit for the size of an integer Floats are limited to 10^{-308} to 10^{308} with 16 to 17 digits of precision
Floating-point literals must contain a decimal point…
Yes: 1200.0 1200.
No: 1200

Strings are delimited (“surrounded”) by single or double quotes: **Examples:** 'Hello World!' "Hello World!"
Strings must be delimited by the same type of quote characters: **Yes:** 'Hello World!'
No: 'Hello World!"
Strings containing quotes must be delimited with the other quote type:

Yes: "This is John's car" **No:** 'This is John's car' 'We all yelled "Hey John!"' "We all yelled "Hey John!""

Single and double-quoted strings must be contained on one line. Triple-quoted strings, however, can span more than one line. Triple-quoted strings may be denoted with three single quotes, or three double quote characters. These strings are mainly used as docstrings for program documentation.

There are only two literal values for the Boolean type:
Yes: True False
No: true false TRUE FALSE 'True' 'False'

B5. Arithmetic, Relational, and Boolean Operators in Python 543
B5. Arithmetic, Relational, and Boolean Operators in Python

+ Addition

- Subtraction * Multiplication

** Exponentiation / Float division

// Truncated division

% Modulus < Less than > Greater than

<= Less than or equal to
>= Greater than or equal to

== Equal != Not equal

and or
not

logical AND logical OR
logical NOT

cond1 = True cond2 = False
2 + 3 - 1 4 2 < 3 True cond1 and cond2 False

2 * 6 12 3 <= 2 False cond1 and not cond2 True
2 ** 6 64 3 == 2 False cond1 or cond2 True
3 / 2 1.5 3 != 2 True not cond1 or cond2 False
3 // 2 1 'A'<'B' True not(cond1 and cond2) True
3.0 // 2 1.0 'a' < 'B' False (2 < 3) and cond1 True
2025 // 100 20 'Hill' < 'Wu' True 2025 % 100 25

** exponentiation
*, /, //, % multiplication, division, modulus (remainder) +, addition, subtraction

< , <=,… in, not in relational, membership, and identity operators not Boolean
not operator
and Boolean and operator
or Boolean or operator

NOTE: All listed operators associate left-to-right, except ** (which associates right-to-left).
B6. Built-in Types and Functions in Python

int Integers
float Floating point numbers complex Complex numbers bool Boolean values
dict Dictionary

str Strings list Lists tuple Tuples

abs(x)

Returns the absolute value of a number (for arguments of type int, long, and float). set Sets
frozenset Immutable sets

chr(i)
Returns a string of one character whose ASCII code is the integeri. The argument must be in the range [0…255] inclusive, otherwise a ValueError will be raised.

cmp(x, y)
Compare the two objectsx and y and returns a negative value if x<y, zero if x==y,and a positive value ifx>y.

dict(arg)
Creates a new dictionary wherearg is a list of the form[(attr1, value1), (attr2, value2), …].

float(arg)
Returns the absolute value of a number (for arguments of type int, long, and float).

format(value, format_spec)
Creates a formatted string from a provided sequence of format specifiers. (See details in Section B10)

frozenset(arg)
Creates a frozen (immutable) set where arg is a sequence (or other iterable type).

(…continued)

help(arg)
Used in interactive mode in the Python Shell. Displays a help page related to the

string provided inarg. When an argument not provided, starts the built-in help system. **print(arg1, arg2, ...)**

id(object)

Prints arguments arg1,arg2, … on the screen.
Returns the “identity” of an object, which is an integer unique to all other objects, and **range(opt_start, stop, opt_step)**

remains the same during an object’s lifetime.
Creates a list of integer values of a given progression. Ifopt_start not provided, then
input(prompt)

sequence begins at 0. If opt_step not provided, then increments by 1.
Prompts user with providedprompt argument to enter input, and returns the typed input as a string with the newline (\n) character removed. (See details in section B7)
Returns a string which is a printable representation ofarg.
int(arg)
reload(module) Returns integer value of provided argument. Argument may be a string or an integer (in which case, the same integer value is returned).Reloads a given module that has already been loaded (imported). Useful for when changes to modules when in interactive mode.
len(arg)
Returns the length ofarg, where arg may be a string, tuple, list, or dictionary.**round(x, opt_n)**
Returns the floating point value *x* rounded to n decimal places. If n is omitted, decimal place **list(arg)**

returned as 0.

Returns a list with items contained in arg, where arg may be a sequence (or other iterable type). If no argument given, returns an empty list.
Returns a new set with values in arg, where arg is a sequence (or other iterable) type. **max(arg)**
Returns the largest value in a provided string, list or tuple.
Returns a new sorted list with values inarg, where arg is a sequence (or other iterable) type. **min(arg)**

str(arg) Returns the smallest value in a provided string, list or tuple.
Returns a string version ofarg.Ifarg is omitted, returns the empty
string.**open(filename, opt_mode)**

tuple(arg) Opens and creates a file object for filename. Optional
argumentopt_mode either'r' (reading),
'w'
(writing), or
'a'
(appending). If omitted, file opened for reading.
Returns a tuple whose items are the same as in arg, where arg may be a sequence
(or other iterable type). If no argument is given, returns a new empty tuple.
ord(str)
type(arg) Given a character (string of length one), returns the Unicode code point
(character encoding
value) for the character, e.g. ord('a') = 97
Returns the type of a value (object) provided in arg.
(…continued)
pow(x, y)
Returns x to the powery. Equivalent tox**y.
print(arg1, arg2, ...)
Prints argumentsarg1,arg2, … on the screen. (See details in section B7)

range(opt_start, stop, opt_step)
Creates a list of integer values of a given progression. Ifopt_start not provided,
then sequence begins at 0. Ifopt_step not provided, then increments by 1.

repr(arg)
Returns a string which is a printable representation ofarg.

reload(module)
Reloads a given module that has already been loaded (imported). Useful for
when making changes to modules when in interactive mode.

round(x, opt_n)
Returns the floating point value x rounded to n decimal places. If n is omitted,
decimal place returned as 0.

set(arg)

Returns a new set with values in arg, where arg is a sequence (or other iterable type).

sorted(arg)

Returns a new sorted list with values inarg, wherearg is a sequence (or other iterable type).

str(arg)

Returns a string version ofarg.Ifarg is omitted, returns the empty string.

tuple(arg)

Returns a tuple whose items are the same as in arg, where arg may be a sequence (or other iterable type). If no argument is given, returns a new empty tuple.

type(arg)

Returns the type of a value (object) provided inarg.

B7. Standard Input and Output in Python 547

B7. Standard Input and Output in Python

Standard input (sys.stdin) is a data stream used by a program to read data. By default, standard input is from the keyboard.

input(prompt) Theinput function sends (optional) string parameter,prompt, to the standard output (the screen) to prompt the user for input. It then returns the line read from the standard input (the keyboard) as a string, with the trailing newline character removed.

print(value, ..., sep=' ', end='\n', file=sys.stdout) Theprint function sends values to the standard output (the screen). Multiple values may be given, each separated by commas. The displayed values are separated by a blank character and ended with a newline character. Each of the default parameter values, sep,end and file, may be changed as keyword arguments. Standard output (sys.stdout) is a data stream used by a program to write data. By default, standard output goes to the screen.

```
>>> name = input('Enter name: ') Enter name: Audrey Smith >>> name
'Audrey Smith'
>>> age = 38
>>> print('His age is', age)) His age is 38

>>> n = int(input('Enter age: ')) Enter age: 28
>>>n
```

>>> print(1, 2, 3, sep='*') 1*2*3

>>> print(1, 2, 3, end='...') 1 2 3...

The prompt string provided for the functioninput often is ended with a blank character to provide space between the end of the prompt and the typed user input.
To keep the cursor on the same line when callingprint, set default parameter end to the empty string as a keyword argument.

The input function reads and returns any input by the user without generating an error. When the input is converted to another type, such as an integer in (2) above, the type conversion function used may raise an exception. Thus, exception handling is needed to check for invalid input.

B8. General Sequence Operations in Python
len(s)
Returns the length of sequences.
s[i] (*selection*)
Returns the item at index i in sequence s.
s[i:j] (*slice*)
Returns subsequence (“slice”) of elements in s from index i to indexj–1.
s[i:j:k] (*slice with step*)
Returns subsequence (“slice”) of everykth item in s[i:j]**.**
s.count(x)
Returns the number of occurrences of x in sequence s.
s.index(x)
Returns the first occurrence ofx in sequence s.
x in s (*membership*)
Returns True ifx is equal to one of the items in sequence s, otherwise returnsFalse.
x not in s (*membership*)
Returns True ifx is not equal to any of the items in sequence s, otherwise returnsFalse.
s1 + s2 (*concatenation*)
Returns the concatenation of sequence s2 to sequence s1 (of the same type).
n * s (or **s * n)**
Returns n (shallow) copies of sequence s concatenated.

min(s)
Returns the smallest item in sequence s.
max(s)
Returns the largest item in sequence s.
B 9 . String Operations in Python 549
B 9 . String Operations in Python

str.find(arg) / str.index(arg)
Both find and index return the lowest index matching substring provided. i Method find returns -1 if substring not found, methodindex returns ValueError.

str.isalpha(arg) / str.isdigit(arg)
Returns True ifstr non-empty and all characters instr are all letters (isalpha), or all digits (isdigit), otherwise returnsFalse.

str.isidentifier()
Returns True isstr is an identifier in Python, otherwise returnsFalse.

str.islower() / str.isupper() / str.lower() / str.upper() Methodsislower /isupper returnTrue if all letters in str are lower (upper) case, and there is at least one letter instr. Methodslower /upper return a copy ofstr with all letters in lower (upper) case.

str.join(arg)
Returns a string which is the concatenation of all strings inarg, where arg is a sequence (or some other iterable) type. If arg does not contain any strings, a TypeError is raised.

str.partition(arg)
For string separator inarg, returns a 3-tuple containing the substring before the separator in str, the separator itself, and the substring following the separator. If separator not found, returns a 3-tuple containingstr, and two empty strings.

str.replace(arg1, arg2)
Returns a copy ofstr with all occurrences ofarg1 replaced byarg2.

str.split(arg)
Returns a list of words in str using arg as the delimiter string. Ifarg not provided, then whitespace (a blank space) is used as the delimiter.

str.strip(arg)
Returns a copy ofstr with leading and trailing chars contained in string arg removed. If arg not provided, then removes whitespace.

B10. String Formatting in Python
format(*value, format specifier)* Built-in function format creates a formatted string from a provided sequence of format specifiers.
A format string may contain the following format specifiers in the order shown: fill_chr, align, width, precision, type

fill_chr Any character except '{' *align* '<' (left-justified) , '>' (requires a provided align specifier) (right-justified) , '^' (centered)

width An integer (total field width)
precision An integer (number of decimal places to display, rounded) *type* 'd' (base 10), 'f' (fixed point), 'e' (exponential notation)

avg = 1.1275, sales = 143235 factor = 134.10456
>>> print('Average rainfall in April:', format(avg,'.2f'), 'inches') Average rainfall for April: 1.13 inches
>>> print('Yearly Sales $', format(sales, ',')) Yearly Sales $ 143,235
>>> print('Conversion factor:', format(factor, '.2e')) Conversion factor: 1.34e+02
>>> print(format('Date', '^8'), format('Num Sold', '^12') Date Num Sold
>>> print('\n'+ format('Date', '<8') + format('Num Sold', '^12') + '\n'+ format('----', '<8') + format('--------', '^12'))
Date Num Sold

B11. Lists in Python 551
B11. Lists in Python
lst[i] = x
Elementi oflst replaced withx (for any objectx).
lst[i:j] = t
Slice oflst fromi toj-1 replaced with sequence (or other iterable type) t.
del lst[i:j]
Removes slicelst[i:j] fromlst.
lst[i:j:k] = t
Replaces items lst[i:j:k] inlst witht
del lst[i:j:k]
Removes slice lst[i:j:k] fromlst.

s.append(x)
Addsx to the end of sequences.
s.extend(x)
Adds the *contents* of sequence x to the end of sequence s.
s.insert(i, x))
Insertsx at indexi in sequence s.
s.pop(i)
Removes and returnss[i]. When argumenti not provided, removes last item ins.
s.remove(x)
Removes and returns the *first* item in sequence s that equals x. **s.reverse()**
Reverses the items in sequences. **s.sort()**
Sorts sequence s from smallest to largest.
B12. Dictionaries in Python

dict(arg)
Returns a new dictionary initialized with items inarg,wherearg may be a list, tuple or string (or other iterable type). If no argument provided, returns an empty dictionary.

len(d)
Returns the number of items in dictionaryd.
d[key]
Returns item of dictionary d with keykey.
d[key] = value Setsd[key] to value.
del d[key]
Removes d[key] from dictionaryd.
key in d
Returns True ifd has key key, otherwise returns False.
key not in d
Returns False ifd has key key, otherwise returns True.
d.clear()
Removes all items from dictionary.
d.get(key)
Returns the value for key in dictionaryd.
d.pop(key)
Removes key/value pair ind for key and returns the associated value.
d.popitem()
Removes and returns an arbitrary key/value pair from dictionaryd.
B13. Sets in Python 553

B13. Sets in Python

set(arg) / frozenset(arg)
Returns a new set or frozenset with items provided in arg, where arg is a sequence (or other iterable type).

len(s)
Returns the number of items in sets.
x in s (x not in s)
Returns True (False) ifs has item equal tox, otherwise returnsFalse (True).
s.add(x) /s.remove(x),s.pop(), s.clear() (*Set type only*) Adds / removesx, Removes-returns arbitrary item froms, Removes all items froms.

s.isdisjoint(other)
Returns True if sets had no items in common with the set provided by argumentother, otherwise returns False.

s.issubset(other) (also **set <= other**)
Returns True if every item ins is also in setother, otherwise returnsFalse.
s.issuperset(other) (also **set >= other**)
Returns True if every item inother is also in sets, otherwise returnsFalse.
set < other (**set > other**)
Returns True ifs is a proper subset (superset) of setother
s.union(other, ...)
Returns a new set containing all items in set s and all other provided sets.
s.intersection(other, ...)
Returns a new set containing all items common to set s and all other provided sets.
s.difference(other, ...)
Returns a new set containing all items in set s that are not also inother provided sets. **s.symmetric_difference(other)**
Returns a new set containing items that are in set s or setother, but not both.
B1 4 . if Statements in Python

if *condition*: *statements*
elif *condition*: *statements*
else:
statements

An if statement may have zero or more elif clauses, optionally followed by anelse clause. As soon as the condition of a given clause is found true, that clause's statements are executed, and the rest of the clauses are skipped. If none of the clauses are found true, then the statements of the else clause are executed (if present). As with all data structures, if statements may be nested.

```
if n == 0: print('n is zero')

if n == 0:
print('n is zero')
else:
print('n is non-zero')

if n < 0:
print('n is less than 0')
elif n == 0:
print('n is zero')
else:
print('nis greater than zero') if n1 == 0:
print('n1 is zero') if n2 == 0:
print('n2 is zero')

if n1 == 0:
print('n1 is zero')
if n2 == 0:

print('n2 is zero') else:
print 'n2 is not zero' else:
print 'n1 is not zero'
```

It is not required to include elif or else clauses in if statements.
When selecting among a set mutually exclusive conditional expressions, the if/elif (with optional else) form ofif statement should be used.

Forgetting to add a semicolon (:) after each conditional expression, and after keywordelse.
2. Improper operator use when checking for equality (e.g., “==”, not “=”).
3. Improper Boolean expression evaluation resulting from unconsidered operator precedence.
4. Improper indentation of nestedif/elif/else clause(s) resulting in faulty logic.

B1 5 . for Statements in Python 555
B1 5 for Statements in Python
for k in *sequence*: *statements*

The for statement is used to control a loop that iterates once for each element in a specified sequence of elements such as a list, string or other iterable type.

empty_str = '', space = ' ' for num in [2, 4, 6, 8]: print(num, end=space)

2 4 6 8
lst = [2, 4, 6, 8]
for k in range(len(lst)):

lst[k] = lst[k] + 1
for num in range(2, 10, 2): print(num, end=space) 2 4 6 8
for num in range(10, 2, -2): print(num, end=space) 10864
for char in 'Hello':
print(char, end=empty_str) Hello
print(lst) 3 5 7 9
t = [[1, 2, 3],[4, 5, 6]]
for i in range(len(t)):
for j in range(len(t[i])): print(t[i][j], end=space) print()
123

When the elements of a sequence need to be accessed but not altered, afor loop that iterates over the values in the sequence is the appropriate approach (1-4). When the elements of a sequence need to be both accessed and updated, a for loop that iterates over the index values in the sequence is the appropriate approach (5,6). Nested for loops can be used to iterate through a list of sequence types (6).

1. Forgetting to add a semicolon (:) after the loop header.
2. Improper indentation of statements contained in the body of the for loop.
3. Forgetting that range(i,j) (andrange(j)) generates values up to, but not including j.

B16. while Statements in Python
while *condition*: Thestatements contained in a while loop body continue to*statements* execute until the condition evaluates toFalse.

empty_str = '', space = ' ' i = 1

while i < 10:
print(i, end=space) i = i + 1

1 2 3 4 5 6 7 8 9 10
i = 10

while i != 0:
print(i, end=space) i = i – 1

10 9 8 7 6 5 4 3 2 1 while True:
print('This loop keeps running!')

This loop keeps running! This loop keeps running! etc.

file = open('filename.txt', 'r') line = file.readline()
while line != empty_str:

line = file.readline()
num = int(input('Enter a positive number: '))

while num < 0:
print('Your input was invalid.')
num = int(input('Enter a positive number: '))

Awhile loop can be used to implement any kind of loop, although somewhile loops are more conveniently implemented as a for loop (1,2).

An *infinite loop* is caused by a condition that always evaluates toTrue (3).
A loop to execute an indefinite number of times, such as when reading from a file, must be implemented as awhile loop (4,5).
Awhile loop is used when accepting and validating user input (5)

Forgetting to add a semicolon (:) after the conditional expression.
2. Not making progress in the body, thereby creating a loop that never ends (i.e. an infinite loop).
3. Improper indentation of statements contained in the body of the while loop.

B17. Functions in Python 557

B17. Functions in Python

A function is a named group of instructions accomplishing some task. A function is *invoked* (called) by providing its name, followed by a (possibly empty) list of arguments in parentheses. Calls to value-returning functions are expressions that evaluate to the returned function value. Calls to non-value returning functions are effectively statements called for their side effects. (Strictly speaking, they are value-returning functions since they return special value None).

def *name*(*parameters*): *statements*
return *expression*

def *name*(*parameters*): *statements*
num1 = 10, num2 = 25, num3 = 35, list1 = [1, 0, 3, 8, 0] def avg3(n1, n2, n3):
'''Returns rounded avg.'''
returnround((n1 + n2 + n3) / 3)
>>>avg3(num1, num2, num3) 23

def dashLine(len, head=''): empty_str = ''
space = ' '

if head != empty_str:
head = space + title + space
print(head.center(len,'-'))

>>> dashLine(16)

>>> dashLine(16,head='price')
---- price ----
def countdown(from, to):

for i in range(from, to-1, -1): print(i,end='...')
>>> countdown(from=5, to=1) 5...4...3...2...1...
def hello():
print('Hello World!')
>>> hello()
Hello World!
def removeZeros(lst):
for k in range(len(lst)-1, -1, -1): if lst[k] == 0:
del lst[k]

```
>>> list1
[1, 0, 3, 8, 0]
>>> removeZeros(list1)
[1, 3, 8]
```

A triple-quoted string as the first line of a function serves as its docstring (1). Functions may define default arguments (2), and be called with keyword arguments (3). A function may be defined having no parameters (4). Mutable arguments passed to a function can become altered (5).

B18. Classes in Python

class *classname*(*parentclass*):

def __init__(self, *args*): '''docstring'''
.
.

def *methodname*(*args*): '''docstring'''
.
.

def *methodname*(*args*): '''docstring'''

A class consists of a set of methods and instance variables. The instances variables are created in special method init.The init method must have an extra first parameter, by convention named self, that is not passed any arguments when the method is called.

A *docstring* is a single or multi-line string using triple quotes that provides documentation for methods, classes and modules in Python.

class XYCoord(object):

def __init__(self, x, y): self.__x = x
self.__y = y

def getX(self):
return self.__x

def setX(self, x): self.__x = x .

.

def __repr__(self): __init__
__repr__
__str__
__neg__
__add__
__sub__
__mul__
__truediv__ __floordiv__ __mod__
__pow__

return '(' + str(self.__x) + ',' + \ __lt__ str(self.__y) + ')'__le__

__eq__ def __eq__(self, xycoord):__ne__ return self.__x == xycoord.getX() and __gt__ self.__y == xycoord.getY()__ge__

Instance variables beginning with two underscores are treated as private. They are accessible only if the mangled form of the name is used: with a single underscore followed by the class name added to the front. For example, private instance variable __x in classXYCoord becomes_XYCoord__x.

B19. Objects in Python 559
B19. Objects in Python
identifier = *classname*(*args*)
An object is an instance of a class. All values in Python are objects.

(1) loc1 = XYCoord(5,10) Creates a newXYCoord object, its reference assigned toloc1 (2) loc2 = XYCoord(5,10) Creates a newXYCoord object, its reference assigned to loc2 (3) loc1
(4) loc2
(5) id(loc1)
(6) id(loc2)
(7) loc1 == loc2
(8) loc1 is loc2
(9) loc3 = loc1
(10) loc3 == loc1
(11) loc3 is loc1
(12) loc3.setX(10,10)

(13) loc3 == loc1

(5,10) , dereferenced value of identifier loc1
(5,10), dereferenced value of identifier loc2
37349072, reference value of identifier loc1
37419120, reference value of identifier loc2
True, comparison of their dereferenced values False, comparison their reference values
Assigns reference value ofloc1 toloc3
True
True
Changes value of loc3 to(10,10)
False

When assigning an object to an identifier, the reference to the object is assigned, not the object itself. Thus, more than one identifier may reference the same object.
Each newly-created object has a unique id. For variablesvar1 and var2,ifid(var1) equalsid(var2) (or var1 is var2 is True) they are referencing the same object.
Mutable objects (e.g., lists) can be
altered without reassignment:

```
>>> lst = [1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
```

Immutable objects (e.g., strings) cannot be altered without reassignment:

```
>>> str1 = 'Hello'
>>> str1.replace('H', 'J') >>> str1
'Hello'
>>> str1 = str1.replace('H', 'J') >>> str1
'Jello'
```

B20. Exception Handling in Python

try: *statements*
except *ExceptionType*: *statements*
except *ExceptionType: statements*

etc.

Exception handling provides a means for functions and methods to report errors that cannot be corrected locally. In such cases, an *exception* (object) is *raised* that can be *caught* by its *client code* (the code that called it), or the client's client code, etc., until *handled* (i.e. the exception is caught and the error appropriately dealt with). If an exception is thrown back to the top-level code and never caught, then the program terminates displaying the exception type that occurred.

```
def genAsterisks(x): """x an integer in range 1-79."""
if x < 1 or x > 79:
raise ValueError('Must enter 1-79')
return x * '*'
#---- main
valid_input = False

while not valid_input: try:
num = int(input("How Many '*'s?: ")) print(genAsterisks(num))
valid_input = True

except ValueError as errorMesg: print(errorMesg)
```

EOFError
FloatingPointError ImportError
IOError
IndentationError IndexError
KeyError
NameError
OverflowError
RuntimeError
SyntaxError
TypeError
ValueError
ZeroDivisionError

The exceptions built-in module is automatically imported in Python.
Built-in exceptions, raised by the built-in functions/methods, may also be raised by user code. New exceptions may be defined as a subclass of the built-inException class. There may be any number ofexcept clauses for a given try block.

In addition to the except clauses, anelse clause may optionally follow, only executed if the try block does not raise any exceptions. Following that, an optional finally clause, if present, is always executed, regardless of whether an exception had been raised or not. The else and finally clauses are often used for resource allocation, such as closing an open file.

B21. Text Files in Python 561
B21. Text Files in Python

A *text file* is a file containing characters, structured as lines of text. Text files can be directly created and viewed using a text editor. In addition to printable characters, text files also contain *non-printing* newline characters,\n, to denote the end of each line of text.

A file that is open for input can be read from, but not written to.
fileref= open(filename, 'r')
fileref.readline()
fileref.close()

Opens and creates a file object for reading filename. Raises anIOError exception if file not found. Reads next line of file. Returns empty string if at end of file. Includes newline character ('\n') in line read.
Closes file. File can be reopened to read from first line.

filename = input('Enter filename: ') inFile = open(filename, 'r')
Enter filename: testfile.txt Hi,

line = inFile.readline() while line != '':
print(line, end='')
line = inFile.readline() This is a test file. Containing five lines. Including one blank line. >>>

A file that is open for output can be written to, but not read from.

fileref = open(filename, 'w') *fileref.write(s)*
fileref.close()

Opens and creates a file object for writing to filename. Writes string s to file. Does *not* include output of'\n'. Closes file. If not closed, last part of output may be lost.

filename = input('Enter filename: ') outFile = open(filename, 'w')

line = input('Enter line of text:') while line != '':
outFile.write(line + '\n')
line = input('Enter line:')

outFile.close() Enter filename: newfile.txt This is the first entered line. This is the second entered line. This is the last entered line.

newfile.txt
This is the first entered line. This is the second entered line. This is the last entered line.

B22. Modules in Python

A Python module is a file containing Python definitions and statements. The module that is directly executed to start a Python program is called the *main module*. Python provides standard (built-in) modules in the Python Standard Library.

Each module in Python has its own *namespace*: a named context for its set of identifiers. The *fully qualified* name of each identifier in a module is of the form*modulename.identifier*.
import modulename
Makes the namespace ofmodulename available, but not part of, the importing module. All imported identifiers used in the importing module must be fully qualified: import math
print('factorial of 16 = ', math.factorial(16))
from modulename import identifier_1, identifier_2, ... identifier_1,identifier_2, etc. become part of the importing module's namespace: from math import factorial
print('factorial of 16 = ', factorial(16))
from modulename import identifier_1 as identifier_2
identifier_1 becomes part of the importing module's namespace as identifier_2
from math import factorial as fact
print('factorial of 16 = ', fact(16))
from modulename import *
All identifiers of modulename become part of the importing module's namespace (except those beginning with an underscore, which are treated as

private).

```
from math import *
print('factorial of 16 = ', fact(16))
print('area of circle = ', pi*(radius**2)
```

Although the import * form of import is convenient in that the imported identifiers do not have to be fully qualified in the importing module, there is the risk of a name clash. In addition, it is not apparent which identifiers are imported, or which module they are imported from.

C. PYTHON STANDARD LIBRARY MODULES

These pages contain selected modules of the Python Standard Library. For a complete listing, see the official standard library at http://docs.python.org/reference/index.html .

The Module Search Path Python modules may be stored in various locations on a particular system. For this reason, when a module is imported, it must be searched for. The interpreter first searches for a built-in (standard) module with that name. If not found, it then searches for the module in the same directory as the executed program. If still not found, the interpreter searches in a list of directories contained in the variable sys.path, a variable of the built-in module sys. The sys module must be imported to access this variable:

```
.. . import sys
.. .
sys.path
```

['C: \\Python32\\Lib\\idlelib','C:\\Windows\\system32\\python32.zip', 'C:\\Python32\\DLLs','C:\\Python32\\lib','C:\\Python32', 'C:\\Python32\\lib\\site-packages']

The particular value for sys.path depends on your particular Python installation. The dir Built-In Function Built-in function dir can be used to find out the names that a particular module defines. This may be used on any module:

```
.. . import math
.. .
dir(math)
```

['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

Note that in addition to the available mathematical methods, there are three special identifers __doc__, __name__ and __package__ listed. The first two provide the docstring (providing brief documentation of the module's contents) and the module name, respectively:

.. . print(math.__doc__)
This module is always available. It provides access to the mathematical functions defined by the C standard.

.. . print(math.__name__)
math
The __package__special identifier is used for modules that contain submodules, called packages.
C1. The math Module

This module contains a set of commonly-used mathematical functions, including number-theoretic functions (such as factorial); logarithmic and power functions; trigonometric (and hyperbolic) functions; angular conversion functions (degree/radians); and some special functions and constants (includingpi and e). A selected set of function from the math module are presented here.

math.ceil
math.fabs(x)
math.factorial(x) math.floor()
math.fsum(s)
math.modf()
math.trunc(X)

returns the ceiling of x (smallest integer greater than or equal to x). returns the absolute value ofx
returns the factorial ofx
returns the floor ofx (largest integer less thanx).
returns an accurate floating-point sum of values ins (or other iterable). returns

the fractional and integer parts ofx.
returns the truncated value ofs.

math.exp(x)
math.log(x,base) math.sqrt(x)

returns e**x, for natural log basee.
returns logx forbase. Ifbase omitted, returnslog x base e. returns the square root ofx.

math.cos(x) math.sin(x) math.tan(x) math.acos(x) math.asin(x) math.atan(x)

returns cosine of x radians. returns sine ofx radians. returns tangent ofx radians. returns arc cosine ofx radians. returns arc sine ofx radians. returns arc cosine ofx radians.

math.degrees(x) math.radians(x)
returnsx radians to degrees. returns x degrees to radians.
math.pi math.e
mathematical constantpi = 3.141592 mathematical constante = 2.718281
C2. The random Module 565
C2. The random Module
This module provides a pseudorandom number generator using the Mersenne Twister algorithm, allowing users to generate random numbers with near uniform distribution over a long period.

random.random() returns random float value x, where 0 <= x < 1.
random.uniform(*a*, *b*) returns random float value x, where a <= x <= b.
random.randint(*a*, *b*) returns random integer value x, where a <= x <= b.
random.randrange(*start*, *stop*, *step*)
returns random integer value x, where start <= x < stop and x = start + n * step, where n is an integer greater than or equal to zero
random.choice(*seq*) random.shuffle(*seq*) random.sample(*seq*, *k*)

returns random element from sequence seq (must be non-empty). randomly reorders sequence seq in place.
returns list (length k) of unique items randomly chosen fromseq.

list = ['a', 'b', 'c', 'd', 'e']
>>> random.random() 0.8568285775611655

```
>>> random.uniform(1, 10) 9.71538746497116
>>> random.randrange(0, 10, 2) 6
>>> random.randrange(0, 10, 2) 8
>>> random.randrange(0, 10, 2) 2

>>> random.randint(1, 10) 6
>>> random.sample(list, 3) ['d', 'c', 'e']

>>> random.choice(list) 'e'
>>> random.shuffle(list) >>> print(list)
['c', 'b', 'e', 'a', 'd']
```

The random module must be imported before it can be used.
The current system time is used to initialize the random number generator. If comparability and reproducibility are important, supply a seed value x by invokingrandom.seed(x) before generating any random numbers.
To generate only even numbers, use the randrange function withstart of 0,step of2.

C3. The turtle Module

This module provides both procedure-oriented and object-oriented ways of controlling *turtle graphics*. A turtle is a graphical entity in an x/y coordinate plane (Turtle screen) that can be controlled in various ways including: its shape, size, color, position, movement (relative and absolute), speed, visibility (show/hide), and drawing status (pen up/pen down).

```
# set screen size
turtle.setup(800,600) # getting the default turtle t = turtle.getturtle()

# get reference to turtle screen screen = turtle.Screen()
# creating a new turtle t = turtle.Turtle()

# set window title bar screen.title('My Turtles')
```

screen.bgcolor(args) ..... background color, specified by name, (RGB) or hex
screen.bgpic(filename) ..... sets GIF file as screen background screen,clear() ..... clears the screen
screen.reset() ..... resets all turtles to their initial state screen.bye() ..... closes turtle screen window

screen.exitonclick() ..... closes turtle screen window on mouse click

showturtle() / hideturtle() ..... makes turtle visible / invisible
turtle.register_shape(filename) .. GIF file name, registers image for use as turtle shape shape(arg) ..... sets turtle to regular or registered shape

turtle.isvisible() ..... returns True if turtle currently visible turtle.position() .....returns current position of turtle as x,y coordinate turtle.towards(x,y) .....returns angle between turtle and coordinate x,y turtle.xcor() .....returns turtle's currentx coordinate turtle.ycor() .....returns turtle's currenty coordinate turtle.heading() .....returns turtle's current heading
turtle.distance(arg) .....returns distance to (x,y) or to another turtle

turtle.pendown() ..... puts turtle's pen down so draws when it moves
turtle.penup() .....lift's turtle's pen so doesn't draw turtle.isdown() .....returns True if pen currently down turtle.pencolor(color) .....sets pen's color by color name or (RGB) turtle.pensize(size) .....sets pen's line thickness, size a positive number turtle.fillcolor(color) .....sets pen's fill color by color name or (RGB) turtle.clear() .....deletes turtle's drawing from screen
turtle.write(arg) .....writes text representation ofarg at turtle's location

turtle.forward(x)/backward(x)... moves turtle forward/backward x pixels
turtle/right(angle)/left(x).....moves turtle left/right byangle
turtle.goto(x,y).....moves turtle to coordinate (x,y) turtle.setx(x)/sety(y) .....moves turtle tox/y locations
turtle.setheading(angle) ....sets heading of turtle byangle turtle.undo() .....undoes last turtle action
tilt(angle).....rotates turtle relative to its current tile-angle
settiltangle(angle) .....sets turtle toangle
tileangle() .....returns the turtle's current tilt angle turtle.speed(speed) .....1-slowest, 2-faster, … 10-fast, 0-fastest turtle.home() .....moves turtle to original position
turtle.circle(radius) .....draws a circle if sizeradius
turtle.dot(size, color) .....draws a dot with given size and color turtle.stamp() .....stamps turtle shape screen, return unique stamp id turtle.clearstamp(id) .....clears turtle stamp with providedid turtle.clearstamps(n) .....clears lastn

stamps, if argument omitted, clears all

turtle.onclick(func) ..... func a function of two arguments called with the x,y coordinates of the location of mouse click
turtle.onrelease(func) ..... func a function of two arguments called with the x,y coordinates of the location of mouse release
turtle.ondrag(func) ..... func a function of two arguments called with the x,y coordinates of the location of mouse click when mouse dragged
turtle.mainloop() .....starts event loop. Must be last statement in a turtle graphics program

C4. The webbrowser Module

This module provides functionality to open Web-based documents in a browser from within a Python program. The module launches the system's default browser to open the target URL, unless a different (supported) browser is specified.

webbrowser.open(***url, new=0, autoraise=True*****)**
Opens the page at the specifiedurl in a browser window. If possible, a value of 0 for new opens the page in an existing browser window, 1 in a new browser window, and 2 in a new browser "tab." Whenautoraise is passed as True, the browser window is raised to the top of the stack (this may happen on some systems regardless ofautoraise value).

webbrowser.open_new(url)/webbrowser.open_new_tab(url) Always opens the specifiedurl in a new browser window or tab, respectively.

webbrowser.get(name)
Returns a controller object for the specified browser type name, allowing the user to use any browser that is registered with or predefined in the module.

```
import webbrowser
webbrowser.open('http://docs.python.org/, 0, True')
import webbrowser
webbrowser.open_new('http://docs.python.org/')
```

```
import webbrowser
browser = webbrowser.get('firefox') browser.open('http://docs.python.org/')
```

Python scripts can be written to generate HTML code, which can then be opened in a browser as part of the program using this module.

This module is not limited to opening Web pages. On some platforms, it can also be used to open documents and media files. Instead of passing an http URL toopen, pass the address of the target file (e.g.file://*host*/*path* or, on local machine,file:///*path*). The functionalities of this module are also available through the command line.

Index

A

569

B C D E
F
G H
I
J K M
L N
O
P Q R
S
T U V
W
X
Y
Z

A Workbench for Inventing and Sharing Digital Forensic Technology
Chet Hosmer Technical Editor: Gary C. Kessler

AMSTERDAM • BOSTON • HEIDELBERG • LONDON NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Syngress is an Imprint of Elsevier

Acquiring Editor: Steve Elliot
Editorial Project Manager: Benjamin Rearick Project Manager: Priya

Kumaraguruparan Designer: Mark Rogers

Syngress is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA
Copyright# 2014 Elsevier Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described here in. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data Application Submitted
British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library
ISBN: 978-0-12-418676-7
For information on all Syngress publications, visit our website at
store.elsevier.com/syngress

Printed and bound in the United States of America 14 15 16 17 18 10 987 65432 1
To my wife Janet, for your love, kindness, patience, and inspiration that you give every day. I am the luckiest guy in the world.

Acknowledgments

My sincere thanks go to:

Dr. Gary Kessler, the technical editor for this book. Gary, your insights, fresh perspective, deep technical understanding, and guidance added great value to the book. Your constant encouragement and friendship made the process enjoyable.

Ben Rearick and Steve Elliot at Elsevier, for your enthusiasm for this topic and all the guidance and support along the way. This spirit helped more than you can know.
The many teachers that I have had over the years in software development and forensics that have helped shape the content of this book. Ron Stevens, Tom Hurbanek, Mike Duren, Allen Guillen, Rhonda Caracappa, Russ Rogers, Jordon Jacobs, Tony Reyes, Amber Schroader, and Greg Kipper.
Joe Giordano, who had the vision in 1998 to create the first U.S. Air Force research contract to study forensic information warfare. This one contract was the catalyst for many new companies, novel innovations in the field, the establishment of the digital forensic research workshop (DFRWS), and the computer forensic research and development center at Utica College. You are a true pioneer.

Endorsements

“Not only does Hosmer provide an outstanding Python forensics guide for all levels of forensics analysis, but also he insightfully illustrates the foundation of a rich collaborative environment that significantly advances the forensic capabilities of the individual, organization, and forensic community as a whole. For analysts, investigators, managers, researchers, academics, and anyone else with an interest in digital forensics: this is a must read!”

Michael Duren (CISSP), Founder of Cyber Moxie

“With today’s rapid changes in technology digital forensics tools and practices are being forced to change quickly just to remain partially effective; and the technical skills investigators relied on yesterday are quickly becoming obsolete. However, with new technology comes new tools and methods, and the Python language is in one of the best possible positions to be leveraged by investigators. Python Forensicsis quite simply a book that is ahead of its time, and because of this, it is the perfect book for both the beginner and the experienced investigator. Chet Hosmer does a great job of helping the reader refresh older skills and create new ones by offering step-by-step instructions and intelligently framing the information for maximum understanding and contextual awareness. The skills you will learn from Python Forensics will help you develop a flexible and innovative toolkit that will be usable for years to come.”

Greg Kipper, Senior Security Architect and Strategist at Verizon

“This book presents a refreshing, realistic view on the use of Python within modern, digital forensics; including valuable insight into the strengths and weaknesses of the language that every knowledgeable forensics investigator should understand.”

Russ Rogers, President of Peak Security, Inc.

“This book is extremely useful for the forensic Python programmer also for those with little or no programming experience, and an excellent reference cookbook for the experienced programmer. The book considers issues relating to Daubert including testing and validation which is vital for the accreditation of forensic solutions.”

Zeno Geradts, Senior Forensic Scientist and R&D coordinator at the Netherlands Forensic Institute

“As always, Chet Hosmer provides a comprehensive and groundbreaking evaluation of a contemporary platform applicable to digital forensics. Extremely well written and user friendly, the book provides a solid foundation for all levels of forensic Python programmers, and includes a much-needed discussion on empirical validation. Quite simply, the book is a must have for all who maintain a digital forensics library.”

Dr. Marjie T. Britz, Clemson University

List of figures

Figure 1.1 Narrowing the gap 3
Figure 1.2 The future digital crime scene 4
Figure 1.3 Data vs. semantics 5
Figure 1.4 The next-generation cyber warrior 5
Figure 1.5 Programming language popularity according to codeview.com 7
Figure 1.6 Test-then code-then validate 7
Figure 2.1 The Python Programming Language Official Web site 17
Figure 2.2 Downloading the Windows installer 18
Figure 2.3 Windows download confirmation 18
Figure 2.4 Executing the Python 2.7.5 Installer 19
Figure 2.5 Python Installation user selection 20
Figure 2.6 Python Installation directory 20
Figure 2.7 Python customization user manual 21
Figure 2.8 TCL/TK install when needed 21
Figure 2.9 Windows user account control 22
Figure 2.10 Successful installation of Python 2.7.5 22
Figure 2.11 Python directory snapshot 23
Figure 2.12 Windows taskbar with Python icon 23
Figure 2.13 Python startup prompt and messages 24
Figure 2.14 Python Hello World 24
Figure 2.15 Windows execution of hashPrint.py 26
Figure 2.16 Ubuntu Linux execution of hashPrint.py 27
Figure 2.17 Python Shell session using hex() and bin() 27
Figure 2.18 Python Shell session entering hex values 28
Figure 2.19 Python Shell creating lists using the range() built-in Standard Library function 28
Figure 2.20 ipRange execution 29
Figure 2.21 Built-in True and False constants 31
Figure 2.22 Python is not a strongly typed language 31
Figure 2.23 Apply an Exclusive OR (XOR) 33
Figure 2.24 Example using the os module from the Standard Library 35
Figure 2.25 Python IDLE integrated development environment 38
Figure 2.26 Snapshot of WingIDE 4.1 Personal 40
Figure 2.27 WingIDE Python Shell display 40
Figure 2.28 WingIDE in action 41
Figure 2.29 Using WingIDE to step through code 42

Figure 2.30 WingIDE examination of the completed list 43
Figure 2.31 WingIDE auto complete feature 43
Figure 2.32 Ubuntu download Web page 12.04 LTS 44
Figure 2.33 Ubuntu terminal window Python command 45
Figure 2.34 Ubuntu software center 45
Figure 2.35 WingIDE running on Ubuntu 12.04 LTS 46

Figure 2.36 Python Shell running on iOS 47
Figure 2.37 iOS implementation of HashPrint 48
Figure 2.38 Apple App Store page regarding Python for iOS 48
Figure 2.39 Windows 8 Phone screenshot of PyConsole launching 49
Figure 2.40 Python Console “Hello World” on a Windows 8 Phone 49
Figure 2.41 Windows 8 Phone HashPrint application execution 50
Figure 2.42 Python Console Windows App Store page 50
Figure 3.1 Cryptographic SmartCard 54
Figure 3.2 Context diagram: python-file system hashing (p-fish) 61
Figure 3.3 p-fish internal structure 62
Figure 3.4 p-fish WingIDE setup 64
Figure 3.5 Demonstration of ParseCommandLine 70
Figure 3.6 pfish –h command 70
Figure 3.7 Test run of pfish.py 84
Figure 3.8 Result directory after pfish execution 84
Figure 3.9 Examining the Result File with Microsoft Excel 85
Figure 3.10 Contents of the pFishLog file 86
Figure 3.11 Linux Command Line Execution 87
Figure 3.12 Linux Execution Results pfish Result File 88
Figure 3.13 Linux Execution Results pFishLog File 89
Figure 4.1 Snapshot of stackdata displaying the baTarget object and the size in bytes of the baTarget bytearray 95
Figure 4.2 p-search context diagram 97
Figure 4.3 p-search internal structure 98
Figure 4.4 WingIDE p-search execution 99
Figure 4.5 p-search execution using only the -h or help option 102
Figure 4.6 Execution test directory for p-search 108
Figure 4.7 Keyword file dump 108
Figure 4.8 p-search sample execution 109
Figure 4.9 Log file contents post execution 109

Figure 4.10 Execution of p-search running on Ubuntu Linux 12.04 LTS 110
Figure 4.11 Execution of p-search running on iMac 111
Figure 4.12 Diagram of word weighting approach 111
Figure 4.13 Weighted characteristics illustration 112
Figure 4.14 p-search execution with indexing capability 115
Figure 5.1 Downloading Python Image Library for Windows 127
Figure 5.2 Windows installation Wizard for the Python Image Library 128
Figure 5.3 Installing Python Image Library on Ubuntu 12.04 LTS 129
Figure 5.4 Internet Photo Cat.jpg 135
Figure 5.5 Map of GPS coordinates extracted From Cat.jpg 138
Figure 5.6 p-gpsExtractor context diagram 140
Figure 5.7 WingIDE Project Overview 140
Figure 5.8 p-gpsExtractor.py execution 159
Figure 5.9 Mapping the coordinates extracted from photos 160
Figure 5.10 Map zoom into Western Europe 161

Figure 5.11 Map zoom to street level in Germany 162
Figure 5.12 Snapshot of Results.csv file 162
Figure 5.13 Snapshot of the Forensic Log file 163
Figure 6.1 John Harrison H1 clock 166
Figure 6.2 A very brief history of time 168
Figure 6.3 Python ntplib download page 174
Figure 6.4 Download of ntplib-0.3.1.tar.gz 175
Figure 6.5 Decompressed ntplib-0.3.1 176
Figure 6.6 Install ntplib 176
Figure 6.7 Verifying the installation 177
Figure 6.8 dir(ntplib) results 177
Figure 6.9 Partial list of NIST time servers 178
Figure 6.10 European NTP Pool Project 178
Figure 7.1 NLTK.org Installation url 185
Figure 8.1 Simplest local area network 206
Figure 8.2 Isolated localhost loopback 208
Figure 8.3 server.py/client.py program execution 211
Figure 8.4 Photo of the actual USS Dallas Los Angeles-class nuclear-powered attack submarine 212
Figure 8.5 Command line launch of the guiPing.py as root 218
Figure 8.6 GUI interface for Ping Sweep 218

Figure 8.7 Ping Sweep execution 224
Figure 8.8 Error handling for misconfigured host range 225
Figure 8.9 Port Scanner GUI 228
Figure 8.10 Port Scanner program launch 228
Figure 8.11 Port Scanner execution with Display All selected 234
Figure 8.12 Port Scanner execution with Display NOT selected 234
Figure 9.1 SPAN port diagram 239
Figure 9.2 SPAN port connections 239
Figure 9.3 Raw TCP/IP packet contents 242
Figure 9.4 Typical IPv4 packet header 243
Figure 9.5 Typical TCP packet header 246
Figure 9.6 Typical UDP packet header 248
Figure 9.7 WingIDE environment for the PSNMT application 250
Figure 9.8 psnmt TCP sample run 259
Figure 9.9 psnmt UDP sample run 260
Figure 9.10 Sample TCP output file shown in Excel 261
Figure 9.11 Sample UDP output file shown in Excel 262
Figure 10.1 Plaintext Rainbow Table output abridged 287
Figure 11.1 Typical cloud configuration 290
Figure 11.2 Cloud execution from iPad 291
Figure 11.3 Desktop execution of the simple and multiprocessing
Python applications executing in the cloud 291
Figure 11.4 Python Anywhere Home Page 292
Figure 11.5 Python Anywhere Plans 293

Figure 11.6 PICloud Home Page 293
Figure 11.7 PICloud Plans 294
Figure 11.8 Digital Ocean Home Page 294
Figure 11.9 Digital Ocean Plans 294
Figure 11.10 Python Anywhere Single Core Execution Results 299
Figure 11.11 Python Anywhere Multi-Core Execution Results 299
Figure 11.12 Standalone Linux Single/Multi-Core Execution Results 299
Figure 11.13 elPassword 8-character combinations of lowercase letters 300
Figure 11.14 elPassword 8-character full ASCII character set 301
Figure 11.15 Last Bit calculation lowercase using 1 computer 301
Figure 11.16 Last Bit calculation lowercase using 100 computers 302
Figure 11.17 Last Bit calculation ASCII set using 100 computers 302

Figure 11.18 Last Bit calculation ASCII set using 10,000 computers 302
Figure 12.1 Multiprocessing in the Cloud 311
Figure 12.2 AMD 6300 Series 16 Core Processor 311
Figure 12.3 Intel Xeon E7 Series 10 Core 20 Thread Processor 311

About the Author

Chet Hosmer is a Founder and Chief Scientist at WetStone Technologies, Inc. Chet has been researching and developing technology and training surrounding forensics, digital investigation, and steganography for over two decades. He has made numerous appearances to discuss emerging cyber threats including National Public Radio's Kojo Nnamdi show, ABC's Primetime Thursday, NHK Japan, Crime Crime TechTV, and ABC News Australia. He has also been a frequent contributor to technical and news stories relating to cyber security and forensics and has been interviewed and quoted by IEEE, The New York Times, The Washington Post, Government Computer News, Salon.com, and Wired Magazine.

Chet also serves as a Visiting Professor at Utica College where he teaches in the Cybersecurity Graduate program. He is also an Adjunct Faculty member at Champlain College in the Masters of Science in Digital Forensic Science Program. Chet delivers keynote and plenary talks on various cyber security related topics around the world each year.

About the Technical Editor

Gary C. Kessler, Ph.D., CCE, CCFP, CISSP, is an Associate Professor of Homeland Security at Embry-Riddle Aeronautical University, a member of the North Florida Internet Crimes Against Children (ICAC) Task Force, and president and janitor of Gary Kessler Associates, a training and consulting company specializing in computer and network security and digital forensics.

Gary is also a part-time member of the Vermont ICAC. He is the coauthor of two professional texts and over 70 articles, a frequent speaker at regional, national, and international conferences, and past editor-in-chief of the Journal of Digital Forensics, Security and Law. More information about Gary can be found at his Web site, http:// www.garykessler.net.

xxiii

Foreword

On June 16, 2008 a user in the home of 2-year old Caylee Anthony googled for the term fool proof suffocation. A minute later, the same user logged onto the MySpace website with the profile of Casey Anthony. Tragically within months, the police found the decomposed remains of the young girl. Prosecutors charged Casey Anthony with first-degree murder and subsequently tried her 3 years later. The trial lasted 6 months and contained over 400 pieces of unique evidence. Sadly, the details of the computer search never made it to trial. The prosecutor's computer forensic examiner employed a tool to retrieve the browser history results. In use of that tool, the examiner only searched the Internet Explorer browser history and not that of the Firefox browser. The moral of the story is that we are only as good as our tools and our understanding how they work.

In contrast to the failure of the computer forensic examiner, let us consider the most feared military force—The Spartan Army. The strength of the Spartan army relied upon the professionalism of its soldiers. From a young age, the elite warriors learned only one occupation—to wage war. In that profession, they relied heavily on the quality of their weaponry and armor. Instead of weapons being issued to a warrior, it was the responsibility of every Spartan warrior to bring his own weapons and armor to war. Fathers passed these tools to their sons

before entering battle. In the following pages, Chet Hosmer passes down modern tools and weapons. As a forensic investigator, your terrain may include a hard drive's unallocated space instead of the pass of Thermopylae. However, just like the Spartan elders, Chet will teach you to forge your own tools. Building your own weaponry is what separates a forensic examiner that makes the gross mistake of missing browser artifacts versus that of a professional examiner.

The following chapters cover a breadth of topics from hashing, keyword searching, metadata, natural language processing, network analysis, and utilizing cloud multiprocessing. Chet covers this range of exciting topics as he teaches you to build your own weapons in the Python programming language. A visiting professor in the Cyber Security Graduate Program at Utica College, Chet is both an educator and a practitioner. He has served as the principal investigator on over 40þ cyber security, digital forensic and information assurance research programs and received international recognition and awards for his work. Like the Spartan elder, his knowledge will hopefully translate to the growth of a new breed of professional. So please enjoy the following pages. And as Spartan wives used to tell their husbands upon entering battle, come back with your shield or on it.

TJ OConnor SANS Red&Blue Team Cyber Guardian xxv

Preface

Over the past 20 years I have had the privilege to work with some of the best, brightest, and dedicated forensic investigators throughout the world. These men and women work tirelessly to find the truth—usually working under less than ideal conditions and under the stress of real deadlines. Whether they are tracking down child predators, criminal organizations, terrorists, or just good old fashion criminals trying to steal your money, these investigators are under the gun and need the best of the best at their fingertips.

I communicate regularly with industry leaders developing the latest forensic products, while evolving their current software baseline to meet the needs of the broadest audience possible. I also communicate with customers trying to solve real-world problems that require immediate answers to hard questions, while the volume of data holding the answer gets larger by the second.

As a scientist and teacher, I see a thirst from students, law enforcement

personnel, and information technology professionals who possess a burning desire, unique investigative skills, an understanding of the problem, and most importantly innovative ideas pertaining to the problems at hand. However, in many cases they lack the core computer science skills necessary to make a direct contribution to the cause.

The Python programming language along with the global environment that supports it offers a path for new innovation. Most importantly the language opens the door for broad inclusion and participation of free tools and technology that can revolutionize the collection, processing, analysis, and reasoning surrounding forensic evidence. This book provides a broad set of examples that are accessible by those with zero or little knowledge of programming, as well as those with solid developer skills that want to explore, jump start, and participate in the expanded use of Python in the forensic domain. I encourage you to participate, share your knowledge, apply your enthusiasm, and help us advance this cause.

INTENDED AUDIENCE

I have written the book to be accessible by anyone who has a desire to learn how to leverage the Python language to forensic and digital investigation problems. I always thought of this as an on-ramp and a beginning that I hope this will inspire you to create something great and share it with the world.

PREREQUISITES

Access to a computer, familiarity with an operating system (Windows, Linux, or Mac) and access to the Internet, coupled with a desire to learn.

READING THIS BOOK

The book is organized with the first two chapters focused on introductory material and setting up the free Python development environment. Chapters 3 through 11 focus on differing problems or challenges within digital investigation, and provide guided solutions along with reference implementations that focus on the core issues presented. I encourage you to use, expand, evolve, and improve the solutions provided. Finally, Chapter 12 looks back and then forward to consider the path ahead.

SUPPORTED PLATFORMS

All the examples in the book are written in Python 2.7.x in order to provide the greatest platform compatibility. The associated web site has solutions for both Python 2.7.x and 3.x whenever possible. As more third party libraries complete support for Python 3.x, all the examples will be available for 2.7.x and 3.x. Most of the examples have been tested on Windows, Linux, and Mac operating systems and will most likely work correctly on other environments that fully support at least Python 2.7.x

DOWNLOAD SOFTWARE
Those purchasing the book will also have access to the source code examples in the book (in both 2.7.x and 3.x—when possible) from the python-forensics.org web site.
COMMENTS, QUESTIONS, AND CONTRIBUTIONS

I encourage you to contribute in a positive way to this initiative. Your questions, comments, and contributions to the source code library at python-forensics.org will make this a resource available to all.

I challenge you all to share your ideas, knowledge, and experience. CHAPTER

Why Python Forensics? 1

CHAPTER CONTENTS

Introduction
..... 1
Cybercrime Investigation Challenges
..... 2
How can the Python Programming Environment Help Meet these Challenges?
..... 6
Global Support for Python 6
Open Source and Platform Independence 8
Lifecycle Positioning 8
Cost and Barriers to Entry 8
Python and the Daubert Evidence Standard
..... 9
Organization of the Book
..... 10
Chapter Review

..... 10
Summary Questions
..... 11
Additional Resources
..... 11

INTRODUCTION

The Python programming language and environment has proven to be easy to learn and use and is adaptable to virtually any domain or challenge problem. Companies like Google, Dropbox, Disney, Industrial Light and Magic, and YouTube just to mention a handful are using Python within their operations. Additionally, organizations like NASA's Jet Propulsion Lab; the National Weather Service; The Swedish Meteorological and Hydrological Institute (SMHI); and Lawrence Livermore National Laboratories rely on Python to build models, make predictions, run experiments, and control critical operational systems.

Before diving straight in, I am sure you would like a little more information about what I will be covering and how a programming environment like Python matches up with digital investigations. Also, you might be interested to know what you will be learning about, generally what the scope of this book is, and how you can apply the concepts and practical examples presented.

The primary purpose and scope of the book is to show you how Python can be used to address problems and challenges within the cybercrime and digital investigation domain. I will be doing this by using real examples and providing the full source code along with detailed explanations. Thus the book will become a set of reference implementations, a cookbook of sorts, and at the end of the day, will hopefully get you involved in developing your own Python forensic applications.

© 2014 Elsevier Inc. All rights reserved.

I will be presenting the material without any preconceived notion about your programming expertise (or lack thereof). I only expect that you have an interest in using the examples in the book, expanding on them, or developing derivatives that will fit your situation and challenge problems. On the other hand, this is not a how to programming book, many of those exist for Python along with a

plethora of online resources.

So, let us get started by defining just some of the challenges we face in cybercrime and digital investigation. These challenges after all were the catalyst behind the book and have come from the past two decades of working on solutions to assist law enforcement; defense and corporate entities collect and analyze digital evidence.

CYBERCRIME INVESTIGATION CHALLENGES

Some of the challenge problems that we face in cybercrime investigation include: The changing nature of investigations: Much of the work over the past two decades has focused on the postmortem acquisition, search, format, and display of information contained on various types of media. I can clearly remember the phone call I received almost two decades ago from Ron Stevens and Tom Hurbanek at the New York State Police. They were investigating a case that involved a Linux computer and were quite concerned about files and other data that might have been deleted that could be impeding the investigation. At that point no technology existed to extract deleted files or fragments that were buried away inside deleted Linux inodes, although several solutions existed for the Windows platform at the time. We worked together to develop algorithms that eventually became a tool named “extractor” that we provided free to law enforcement.
The move from simply extracting data, recovering deleted files, and scouring unallocated or slack space from computers has rapidly shifted just in the last couple of years. Today we focus most of our attention on smart mobile devices, dynamically changing memory, cloud applications, real-time network forensics, automotive data analysis, and weather-based forensics, just to mention a few. In addition, new work is addressing the association of direct digital forensic evidence with a broad range of

instantly available electronic information. Whether this information comes from text
messages, Facebook posts, tweets, Linkedin associations, metadata embedded in digital photographs or movies, GPS data that tracks our movements or the digital fingerprints left from every Web site we surf, all may be relevant and used in civil
or criminal cases. The question is how do we connect these dots while maintaining
forensic efficacy?

The widening gap between technology developers and investigators: Investigators, examiners, incident response personnel, auditors, compliance experts tend to
come into this field with a background in social science, whereas technology developers tend to have backgrounds in computer science and engineering. Clearly, there are some excellent examples of crossovers in both directions, but the vocabulary, thought process, and approach to problem solving can be quite different. Our goal, as depicted in Figure 1.1, is to leverage Python forensic solutions to close that gap and create a collaborative nonthreatening environment whereby computer
science and social science can come together.

The challenge is to develop a platform, vernacular environment where both social
scientists and computer scientists can comfortably communicate and equally participate in the process of developing new forensic solutions. As you will see, the Python
environment provides a level playing field, or common ground at least, where new
innovations and thought can emerge. This has already shown to be true in other scientific fields like Space Flight, Meteorology, Hydrology, Simulation, Internet Technology advancement, and Experimentation. Python is already providing valuable contributions in these domains.

Cost and availability of new tools: With a couple of exceptions (for example, EnCase® App Central), most new innovations and capabilities that come through vendor channels take time to develop and can add significant cost to the investigator's toolkit. In the past, investigators carried with them just a handful of hardware
and software tools that they used to extract and preserve digital evidence. Today, to
address the wide range of situations they may encounter, 30-40 software

products
may be necessary just to perform acquisition and rudimentary analysis of the digital
crime scene. Of course this is just the start of the investigative process and the number and variety of analytic tools continues to grow.
The true cost and cost of ownership of these technologies can be staggering, especially when you factor in education and training. The barrier to entry into the
field can easily reach high five or even six figures. This is in a field where backlogs
continue to grow at law enforcement agencies around the world. Backlogs are also
growing within the corporate sector, which is dealing with human resource actions,
corporate espionage, insider leaks, and massive amounts of regulatory requirements.

Social science Social science
Python forensics Computer
science
Computer science

Today Future FIGURE 1.1
Narrowing the gap.

FIGURE 1.2
The future digital crime scene.

It is obvious that we need a better onramp and new ways for individuals that have both interest and aptitude, to participate and make entry into the digital investigative field easier and more streamlined. As we move forward in time, the digital crime scene will look more and more like the depiction in Figure 1.2.

Data vs. semantics : Due to constrained resources, new technologies and innovation we must move from simple data analysis and search to rapid semantic understanding and even situational awareness. Investigators need tools that can help them narrow in on targets or hotspots within an investigation, in order to better apply resources and more quickly identify important leads. This is especially true for investigations that are ongoing such as active fraud cases, denial of service attacks, sophisticated malicious code breaches, violent crimes such as murder, child abduction, rape, and aggravated assaults that have a digital component.

In addition, we must capture the knowledge and process of the most seasoned investigators before they leave the field. What is needed is a better way to capture that knowledge and experience, as most of the investigative analysis of digital evidence happens today between the ears of the investigator. To illustrate the difference, Figure 1.3 shows what we mean by data vs. semantics or meaning. The image on the left depicts a GPS coordinate that we typically extract from images, mobile devices, or online transactions. The image on the right is the mapped location—in other words the meaning or translation of the

GPS location into mapped data. These GPS data also include timestamps that allows us to place a device at a particular location at a specific date and time.

The next-generation investigator :We must generate interest in this field of study to attract the best and brightest to a career in cybercrime investigation. In order to do so, this new breed needs to not only use tools and technology but also must play an important role in researching, defining, evaluating, and even developing some of these high demand and sophisticated next-generation capabilities (Figure 1.4).

Lack of a collaboration environment: Cybercriminals have the advantage of untethered collaboration, along with access to resources that assist in the execution

FIGURE 1.3
Data vs. semantics.

FIGURE 1.4
The next-generation cyber warrior.

of distributed attacks and sophisticated cybercrime activities. Investigators and developers of new methods and techniques need that same advantage. They require a platform for collaboration, joint development, and access to new innovations that could be directly applied to the situation at hand.

HOW CAN THE PYTHON PROGRAMMING ENVIRONMENT HELP MEET THESE CHALLENGES?

Creating an environment where social and computer scientists can collaborate and work together is challenging. Creating a platform to develop new technology-based solutions that address the broad range of digital investigation challenges outlined earlier in this chapter is difficult. Doing them both together is a real challenge, and whenever you take on a challenge like this it is important to consider the underpinnings so that you create the best chance for success.

I personally have a few important considerations that have served me well over the years, so I will share them with you here.

1. Does the platform you are building on have broad industry support?
2. Is there an ample supply of technical data regarding the subject along with a large cadre of talent or subject matter experts?
3. Is the technology platform you are considering open or closed?
4. Where does the technology exist along its lifecycle (i.e., too early, too late or mature, and evolving)?
5. What is the cost or other barriers to entry? (especially if you are trying to attract a broad array of participants)
6. Finally, since we are trying to bridge the gap between social and computer science, is the environment well suited for cross-disciplinary collaboration?

Global support for Python
Python was created by Guido van Russom in the late 1980s with the fundamental premise that Python is programming for everyone. This has created a groundswell of support from a broad array of domain-specific researchers, the general software development community, and programmers with varying backgrounds and abilities. The Python language is both general purpose and produces easily readable code that can be understood by nonprogrammers. In addition, due to Python's intrinsic extensibility, copious amount of third-party libraries and modules exist. Numerous web sites provide tips, tricks, coding examples, and training for those needing a deeper dive into the language. It may surprise you that Python was ranked as the number 1 programming language in 2013 by codeeval.com (see Figure 1.5) edging out Java for the first time. A great place to start is at the official Python programming language web sites python.org.

Finally, sophisticated integrated software development environments exist that allow even the novice developer to innovate new ideas and design, and then build

FIGURE 1.5
Programming language popularity according to codeview.com.

and test their inventions and prototypes. Python is an interpreted language; however, compilers are available as well, of course. As shown in Figure 1.6, developers have adopted the test-then code-then validate mindset.

By utilizing the Python Shell, experienced and novice users alike can experiment with the language, libraries, modules, and data structures before attempting to integrate them into a complete program or application. This promotes experimentation with the language, language constructs, objects, experimentation with performance considerations, and libraries, and allows users to explore and tradeoff approaches

FIGURE 1.6
Test-then code-then validate.

prior to putting them into practice. Once confident with the use, characteristics and behaviors of the language, the integration into working programs tends to go more smoothly. In addition, this experimentation often leads to testing considerations that can then be applied to the working programs once they have been completed, thus completing the cycle of test-code-validate.

Open source and platform independence
Since Python is an open source environment, developers continue to create compatible versions that run across multiple platforms. Python implementations exist for today's most popular platforms including Windows, Linux, and Mac OS X as you would expect. However, the support for Python is much broader. Support for mobile device operating systems such as Android, iOS, and Windows 8 is also available. In addition, Python is supported on platforms that you might not expect such as AIX, AS/400, VMS, Solaris, PalmOS, OS/2, and HP-UX, just to mention a few. What this means for cybercrime investigators is portability of these applications for today's platforms, yesterday's platforms, and future platforms.

In March 2013, NVIDIA announced support for Python developers by opening the door to GPU-Accelerated Computing using NVIDIA CUDA, allowing for parallel processing capabilities, and advancing the performance of virtually any Pythondeveloped application. This will deliver the ability to handle and process big data, perform advanced analytical operations, perform deductive and inductive reasoning, and meet future computational challenges. This versatility ensures that the investment made in creating new investigative solutions will be useable and sharable with colleagues that use different flavors of computing platforms.

Lifecycle positioning
Python today sits in the best position possible to be leveraged for investigative applications. The language is mature, hundreds of thousands of developers are experienced, a strong support organization is in place, extensible libraries are plentiful, applications are portable to a wide range of computing platforms, the source code is open and free, and new innovations to the core language are keeping pace with hardware and operating system advancements.

Cost and barriers to entry
One of the keys to Python's success is the lack of virtually any barrier to entry. The development environment is free, the language is platform independent, the code is as easy to read and write as English, and support is vast and worldwide. In my opinion, it should revolutionize the development of new cybercrime, forensic, and incident response-based solutions. The key is to develop outreach that will encourage, attract, and open the doors to social scientists, computer scientists, law enforcement organizations, forensic labs, standards bodies, incident response teams, academics,

students, and virtually anyone with domain expertise within the broadest definition of cybercrime.
PYTHON AND THE DAUBERT EVIDENCE STANDARD

As many of us have encountered, the Daubert standard provides rules of evidence at the U.S. Federal level along with about one-third of the states that deal with the admissibility of expert testimony including scientific data. Digital data collected and analyzed with forensic software employed by an "expert" can be challenged by using, in laymen's terms, a Daubert motion to suppress or challenge the efficacy of the expert and/or the evidence produced by technology utilized.

In 2003, Brian Carrier [Carrier] published a paper that examined rules of evidence standards including Daubert, and compared and contrasted the open source and closed source forensic tools. One of his key conclusions was, "Using the guidelines of the Daubert tests, we have shown that open source tools may more clearly and comprehensively meet the guideline requirements than would closed source tools."

The results are not automatic of course, just because the source is open. Rather,

specific steps must be followed regarding design, development, and validation.

1. Can the program or algorithm be explained? This explanation should be explained in words, not only in code.
2. Has enough information been provided such that thorough tests can be developed to test the program?
3. Have error rates been calculated and validated independently?
4. Has the program been studied and peer reviewed?
5. Has the program been generally accepted by the community?

The real question is how can Python-developed forensic programs meet these standards? Chapters 3–11 comprise the cookbook portion of the book and each example attempts to address the Daubert standard by including:

1. Definition of the Challenge Problem
2. Requirements Definition
3. Test Set Development
4. Design Alternative and Decisions
5. Algorithm Description (English readable)
6. Code Development and Walk-Through
7. Test and Validation Process
8. Error Rate Calculation
9. Community Involvement

This approach will provide two distinct advantages. First, the cookbook examples provided in the text will be useable out of the box and should meet or exceed the rules of evidence standards. Second, the process defined will assist both experienced and novice developers with an approach to developing digital investigation and forensic solutions. The cookbook examples then provide a model or reference implementations using Python that are designed as an educational and practical example.

ORGANIZATION OF THE BOOK

In order to support the broadest audience of potential contributors to new cybercrime investigative technologies, I have arranged the book to be accessible to those with little or no programming experience, as well as for those that want to dive directly into some of the more advanced cookbook solutions.

Chapter 2 will provide a walk-through for those wishing to setup a Python

software environment for the first time. This step-by-step chapter will include environments for Linux and Windows platforms and will include considerations for Python 2.x and Python 3.x. I will also cover the installation and setup of high-quality thirdparty libraries that I will leverage throughout the book, along with integrated development environments that will make it easier to master Python and manage your projects.

Chapter 3 covers the development of a fundamental Python application that will outline one of the most common digital investigation applications—File Hashing. A broad set of one-way hash algorithms that are directly implemented within the core Python distributions will be covered. I will then show how this simple application can be transformed into a more sophisticated cyber security and investigation tool that could be applied immediately.

Each of the Chapters 4–11 tackles a unique cybercrime investigation challenge and delivers a Python cookbook solution that can be freely used, shared, and evolved, and includes opportunities for you to participate in the future expansion.

Chapter 12 takes a look at future opportunities for the application of Python within cybercrime investigation, a broader set of cyber security applications, and examines high-performance hardware acceleration and embedded solutions.

Finally, each chapter includes a summary of topics covered, challenge problems, and review questions making the book suitable for use in college and university academic environments.

CHAPTER REVIEW

In this chapter, we took a look at the challenges that are facing cybercrime investigators, incident response personnel, and forensic examiners that are dealing with a plethora of digital evidence from a multitude of sources. This chapter also discussed the computer science and social science gap that exists between the users of forensic/ investigative technologies and the developers of the current solutions. We examined

the key characteristics of the Python programming environment which make it well suited to address these challenges. These characteristics include the open

source nature of the Python environment, the platform independent operational model, the global support and available technical data, and the current lifecycle positioning. Also, Python-developed solutions (provided they are done right) will meet or exceed the Daubert rules of evidence requirements. Finally, the organization of the book was discussed to give readers a better understanding of what to expect in upcoming chapters.

SUMMARY QUESTIONS

1. What are some of the key challenges that face forensic investigators today and what potential impacts could these challenges pose in the future?
2. From what has been presented or based on your own research or experience, what do you believe are the key benefits that Python could bring to forensic investigators?
3. What other organizations are using Python today for scientific endeavors and how is the use of Python impacting their work?
4. What other software languages or platforms can you think of that are open source, cross-platform, have global support, have a low barrier to entry, are easily understood, and could be used to collaborate among both computer scientists and social scientists?
5. What forensic or investigative applications can you think of that either do not currently exist or are too expensive for you to buy into?

Additional Resources

Open Source Digital Forensic Tools—The Legal Argument. DigitalEvidence.org, http:// www.digital-evidence.org/papers/opensrc_legal.pdfhttp; 2003.
Python Programming Language—Official Website. Python.org, http://www.python.org.
Basu S. Perl vs Python: why the debate is meaningless. The ByeBaker Web site, http://bytebaker. com/2007/01/29/perl-vs-python-why-the-debate-is-meaningless/; 2007 [29.01.07].
Raymond E. Why Python? The Linux Journal 73, http://www.linuxjournal.com/article/3882; 2000 [30.04.03].

CHAPTER

Setting up a Python Forensics Environment 2

CHAPTER CONTENTS

Introduction 14
Setting up a Python Forensics Environment 14
The Right Environment 15
The Python Shell 16
Choosing a Python Version 16
Installing Python on Windows 17
Python Packages and Modules 24
The Python Standard Library 25
What is Included in the Standard Library? 27
Built-in Functions 27
hex() and bin() 27
range() 28
Other Built-in Functions 30
Built-in Constants 31
Built-in Types 32
Built-in Exceptions 33
File and Directory Access 34
Data Compression and Archiving 34
File Formats 35
Cryptographic Services 35
Operating System Services 35
Standard Library Summary 36
Third-Party Packages and Modules 36
The Natural Language Toolkit [NLTK] 36
Twisted Matrix [TWISTED] 37
Integrated Development Environments 37
What are the Options? 37
IDLE 38

WingIDE 39
Python Running on Ubuntu Linux 42
Python on Mobile Devices
..... 46
iOS Python App 46
Windows 8 Phone 49
A Virtual Machine
..... 51 Chapter
Review 51
Summary Questions
..... 51 Looking Ahead
..... 52 Additional
Resources 52

© 2014 Elsevier Inc. All rights reserved.

INTRODUCTION

A couple of decades ago, I was working for a large defense contractor and was part of a team that was developing a secure embedded device. My initial role was to setup a development environment for the team to use. This may seem like a pretty simple task. However, the embedded security hardware was completed and handed off to me, but the device itself did not include an operating system or supporting libraries—it was basically an open slate. Thus, the first painstaking task was to develop a boot loader that would allow me to load a program onto the device. Once that was completed, I needed to develop the ability to interface with the board and load additional software (the operating system, shared libraries, applications, etc.), that would bring the security hardware contained within the device online.

This interface software needed to include a debugger that would allow us to control the operating system and application software being developed by the team while it was running on the device. For example, the ability to start the program, stop the program, inspect variables, registers, etc., perform single steps, and set breakpoints in the code, all this was accomplished through a 19,200 baud RS232 interface.

You might be asking, what does this twentieth century example have to do with Python? The answer is simple, the requirements for a stable feature-rich development environment still exist in the twenty-first century, but now we have

modern tools. Without the proper development environment, the chance of successfully developing high-quality, function-rich forensic or digital investigation software is quite low.

SETTING UP A PYTHON FORENSICS ENVIRONMENT

Many considerations exist before setting up an environment. I will explore some of the key areas that I myself consider when setting up an environment like this. Here are some important considerations:

1. What is the right environment for your situation? Are you a professional software developer; a novice developer with solid investigative skills and ideas that you would like to explore; do you work in a forensic lab or support an incident response team with new tools and methods; or maybe you might work in the IT Security group at a corporation and need better ways to collect and analyze what is happening on your network.

2. How do you choose the right third-party libraries and modules that enhance your program and allow you to focus on your application and not on reinventing the wheel?

3. What is the right integrated development environment (IDE) and what capabilities should be included in that environment?
a. Code intelligence that provides automatic completion, built-in error

indicators source browser, code indices, and fast symbol lookup. b. A robust graphical debugger that allows you to set breakpoints, single-step through code, view data, and examine variables.
c. A powerful programmer's editor that has a complete understanding of the Python language rules, advanced search tools, bookmarking, and code highlighting.
d. Cross-platform support such that you can choose your platform and environment, (i.e., Windows, Linux, or Mac), and select any Python version from 2.x to 3.x and Stackless Python.

Stackless Python is a relatively new concept, allowing Python programs to execute which are not limited by the size of C Stack. In many environments, the size of stack memory is limited in comparison to the amount of heap memory.

Stackless Python utilizes the heap and not the stack which provides greater distributed processing possibilities. For example, this allows for thousands of independently running tasklets to be launched. Online multiuser gaming platforms use this to support thousands of simultaneous users. I am sure if you are thinking ahead you can imagine some interesting digital investigation and forensic applications for such an environment.

e. A unit testing capability that enables you to thoroughly validate your code within common testing frameworks such as unittest, doctest, and nose.
f. Built-in revision control for advanced projects along with direct integration with popular revision control systems such as Mercurial, Bazaar, Git, CVS, and Perforce. When building larger applications that contain many moving components managing the many revisions becomes important. This selection will ultimately determine if the tools and applications developed will meet the standards of quality that are essential ingredients for digital investigation and forensic applications.

It is important right up front to realize this book is about building forensic applications that must meet the Daubert standard, thus we are not just hacking out some code that will work most of the time, but rather code that should work all the time or fail gracefully. And most importantly, we need to develop code that will create admissible evidence.

THE RIGHT ENVIRONMENT

One of the initial choices you will have to make is the platform you intend to use for the development of Python forensic applications. As discussed in Chapter 1, Python and Python programs will execute on a variety of platforms including the latest desktop, mobile, and even legacy systems. However, that does not mean you need to develop your applications on one of those platforms. Instead you will most likely be developing your applications on a Windows, Linux, or Mac platform that supports the latest development tools.

One great thing about Python ... If you follow the rules, develop quality programs and make sure you consider cross-platform idiosyncrasies, no matter what platform you choose for development, the resulting Python programs you create should easily run on a variety of operating systems that have a properly installed Python system.

The Python Shell

The Python Shell delivers an object-oriented, high-level programming language that includes built-in data structures and an extensive Standard Library. Python employs a simple easy to use syntax that virtually anyone can learn, provides support for thirdparty modules and packages which encourages program code reuse and sharing, and is supported on virtually all major platforms and can be freely distributed. What that delivers to the digital investigator or forensic specialist is the ability to quickly develop programs that will augment or even replace current tools and then immediately share them with the community. One of the other benefits of the interpreted environment is the ability to experiment with the Standard Library, third-party modules, commands, functions, and packages without first developing a program. This allows you to ensure that the commands, functions, and modules you are planning to use provide the results and performance you are looking for. Since there are many

options and sources for these functions with more arriving on the scene every day, the interpreter allows you to easily experiment before you commit to a final approach.

CHOOSING A PYTHON VERSION

As with any programming environment, many currently supported versions of Python are available. However, two basic standards of Python exist today in versions 2.x and 3.x. The change from Python 2.x to 3.x has caused some difficulties in portability, and programs and libraries written for 2.x require modification to work within 3.x.

Some of the core Python functions have changed mainly due to full support of Unicode in version 3.x. This not only affects programs but also affects previous modules that have been developed and have not yet been ported and or validated for Python 3.x. Based on this conundrum, I have decided to develop the examples supplied in this book to conform to the 2.x standard, giving us access to the broadest set of third-party modules and compatibility across more platforms. The source code in the book will be available online as well, whenever possible I will provide both a 2.x and 3.x version of the source.

In addition, the 2.x class of Python has been proven, validated, and deployed in a broad set of applications. Therefore, developing forensic or digital investigation applications using the 2.x version provides us with a solid underpinning and broadest deployment platform. Also, once 3.x becomes broadly embraced and

the third-party libraries become available and certified, we will have all the information necessary to port and even enhance the applications in this book.

Now that we have selected a version to begin with, I will walk you through setting up Python on a Windows desktop.

INSTALLING PYTHON ON WINDOWS

If you Google “Python Installation” you will get a little over 7 million page hits, as of this writing. In my opinion, the best and safest place to obtain tested standard Python installations is at www.python.org, which is the Python Programming Language Official Web site [PYTHON], shown in Figure 2.1. At this page, I selected Python version 2.7.5.

Next, I navigate to the download page and select the appropriate version for my situation. In this case I am going to select:

Python 2.7.5 Windows x86 MSI installer 2.7.5 (sig)

This will download the Windows runtime environment (as shown in Figure 2.2). Also shown is the hash of the download file allowing you to validate the download.

FIGURE 2.1

The Python Programming Language Official Web site.

This is a production release. Please report any bugs you encounter.

We currently support these formats for download:

- XZ compressed source tar ball (2.7.5) (sig)
- Gzipped source tar ball (2.7.5) (sig)
- Bzipped source tar ball (2.7.5) (sig)
- Windows x86 MSI Installer (2.7.5) (sig)
- Windows x86 MSI program database (2.7.5) (sig)
- Windows X86-64 MSI Installer (2.7.5) [1] (sig)
- Windows X86-64 program database (2.7.5) [1] (sig)
- Mac OS X 64-bit/32-bit x86-64/i386 Installer (2.7.5) for Mac OS X 10.6 and later [2] (sig). [You may need an updated Tcl/Tk install to run IDLE or use Tkinter, see note 2 for instructions.]
- Mac OS X 32-bit i386/PPC Installer (2.7.5) for Mac OS X 10.3 and later [2] (sig).

The source tarballs are signed with Benjamin Peterson's key (fingerprint: 12EF 3DC3 8047 DA38 2D18 A5B9 99CD EA9D A413 5B38). The Windows installer was signed by Martin von Löwis' public key, which has a key id of 7D9DC8D2. The Mac installers were signed with Ned Deily's key, which has a key id of 6F5E1540. The public keys are located on the download page.

MD5 checksums and sizes of the released files:

```
b4f01a1d0ba0b46b05c73b2ac909b1df  14492759  Python-2.7.5.tgz
6334b666b7ff2038c761d7b27ba699c1  12147710  Python-2.7.5.tar.bz2
5eea8462f69ab1369d32f9c4cd6272ab  10252148  Python-2.7.5.tar.xz
e632ba7c34b922e4485667e332096999  18236462  python-2.7.5-pdb.zip
55co56948dcee3afb53f65d8bb425f20  17556546  python-2.7.5.amd64-pdb.zip
83f5d9ba639bd2e33d104df9ea969f31  16617472  python-2.7.5.amd64.msi
0006d6219160ce6abe711a71c835ebb0  16228352  python-2.7.5.msi
ead4f83ec7823325ae287295193644a7  20395084  python-2.7.5-macosx10.3.dmg
248ec7d77220ec6c770a23df3cb537bc  19979778  python-2.7.5-macosx10.6.dmg
```

[1] *(1, 2)* The binaries for AMD64 will also work on processors that implement the Intel 64 architecture (formerly EM64T), i.e. the architecture that Microsoft calls x64, and AMD called x86-64 before calling it AMD64. They will not work on Intel Itanium Processors (formerly IA-64).

[2] *(1, 2)* There is important information about IDLE, Tkinter, and Tcl/Tk on Mac OS X here.

FIGURE 2.2
Downloading the Windows installer.

As you would expect, selecting this link presents you with a Windows dialog box as shown in Figure 2.3, confirming that you wish to save the installation file. Selecting OK will download the file and save it in your default download directory.

Examining the contents of my download directory, you can see that I have actually downloaded both the latest 2.x and 3.x releases. I am now going to select and execute the 2.7.5 Installer (Figure 2.4).

FIGURE 2.3
Windows download confirmation.

FIGURE 2.4
Executing the Python 2.7.5 Installer.

Upon execution, the Installer prompts you to determine whether Python should be installed for the current user or for all users on this machine. This is your choice of course, but if you choose to install this for every user make sure they are trusted and know the risks. The Python environment runs at a high privilege level and has access to operating system functions that not everyone should necessarily need (Figure 2.5).

Python allows you to select where the environment will be installed. The default location for Python 2.7.5 is C:\Python27, although you can specify another location (Figure 2.6). You should make a note of this so you can inspect and reference the directories and files stored there. This will be especially helpful later when you are looking for certain libraries, modules, tools, and documentation.

I have decided to customize my installation slightly. I want to ensure that the Python documentation is stored on my local hard drive so I can have access anytime. Also, I do not expect to need the TCL/TK Graphical User Interface (GUI) module, so I have selected this to only be installed when needed; this will reduce the size of the installation. If you have no restriction on disk storage, including this with your install is not an issue. We will be installing other easier to use GUI packages and modules later in the book (Figures 2.7 and 2.8).

Next, Windows will display the typical User Account Control (UAC) to help you stay in control of your computer by informing you when a program makes a change that requires administrator-level permission (Figure 2.9). Windows will also verify the digital signature of Python and display the name of the organization associated with the certificate used to sign the installer. Selecting Yes will install Python and allow the appropriate system changes to be made, provided that the user has administrative privilege. You might notice that this screenshot was taken with my iPhone because during UAC acknowledgment Windows locks out all other program access until either Yes or No is selected.

FIGURE 2.5 Python Installation user selection.

FIGURE 2.6
Python Installation directory.

FIGURE 2.7
Python customization user manual.

FIGURE 2.8
TCL/TK install when needed.

FIGURE 2.9
Windows user account control.

Finally, Python has been installed and displays the successful completion of the installation as shown in Figure 2.10.
We can now navigate to the C:\Python27 directory (or the path name you specified for Python installation) and examine the contents. As you can see in Figure 2.11,

FIGURE 2.10
Successful installation of Python 2.7.5.

FIGURE 2.11
Python directory snapshot.

there are folders that contain Docs, Lib, DLL, and Tools along with key files License, News, and a Readme. Most importantly there are two application files Python and pythonw. These are the Python executable files.

To utilize the Python interactive interpreter you would double click on Python.If you have an existing Python program, and you do not want to display the interpreter window you would execute pythonw along with the associated Python file to execute (more on this later). For our immediate use, we will be using the Python application file. For easy access you can add this application to your Windows Taskbar and you will see the Python icon as highlighted in Figure 2.12.

By launching the Python Taskbar icon (clicking it), the Python Interactive window is displayed as shown in Figure 2.13. Your window will look a bit different from mine, as I have set mine up to be black on white to provide easier book publication. The title bar shows the directory the application was started from C:\Python27. The initial text lists the version of Python in this case Python 2.7.5, the date of the build and information regarding the processor, and Windows version in this case win32. The next line provides some helpful commands such as license, credits, help, and the copyright message. The next line starting with>>>is the Python prompt awaiting instructions. The tradition

here would be of course to test the installation by writing the standard programming language Hello World program, which only takes a single

FIGURE 2.12
Windows taskbar with Python icon.

FIGURE 2.13
Python startup prompt and messages.

FIGURE 2.14
Python Hello World.

line in Python as shown in Figure 2.14. I added a second print statement as well, promoting the idea of open and free forensic software that can be developed in Python, without the need for a HASP.

For those unfamiliar with a HASP (Hardware Against Software Piracy) (sometimes referred to as a Dongle) this is a device that provides copy protection for certain software programs. Without the HASP inserted, the software will not operate, thus only allowing licensed users to utilize the software on a single machine or network.

Now that we have at least verified that the Python interpreter is running and

accepting commands let us take a look at the commands, language structure, packages, and modules that are available.

PYTHON PACKAGES AND MODULES

Adding core functionality to an existing programming language is a standard part of software development. As new methods and innovations are made, developers supply these functional building blocks as modules or packages. Within the Python network, the majority of these modules and packages are free, with many including the full source code, allowing you to enhance the behavior of the supplied modules and to independently validate the code. Before we jump into adding third-party modules to Python, we need to understand what comes standard out of the box or more

specifically what is included in the Python Standard Library. I think you will be pleasantly surprised at the breadth of capability delivered as part of the Standard Library along with the built-in language itself.

The Python Standard Library

Python's Standard Library [Standard Library] is quite extensive and offers a broad range of built-in capabilities. These built-in functions are written primarily in the C programming language for both speed and abstraction. Since the Standard Library layer is compatible across systems, interfacing with platform-specific APIs (application programming interfaces) are abstracted or normalized for the Python programmer.

One of the fundamental operations that investigators perform is the generation of one-way cryptographic hash values.

A one-way cryptographic hash is used to create a signature of an existing string of bytes regardless of length (typically referred to as a message digest). One-way hashes have four basic characteristics: (1) A function that easily calculates and generates a message digest. (2) Possessing the message digest value alone provides no clues as to the original message or file. (3) It is infeasible (or computational difficult) to change the contents of the message or file without changing the associated message digest. (4) It is infeasible to find two messages or files that differ in content, but produce the same message digest. It should be noted that attacks against known hash methods like MD5 and SHA-1 have been

successful under certain controlled situations.

The Python Standard Library contains a built-in module named hashlib that can perform one-way cryptographic hashing. The following simple hashing example can be executed on virtually any Python platform (Windows, Linux, Mac, iOS, Windows 8 Phone, Android, etc.) and will produce the same result.

```
#
# Python forensics
# Simple program to generate the SHA-256 # one-way cryptographic hash of a given string

# Step 1
# Instruct the interpreter to import the # Standard library module hashlib

import hashlib
# print a message to the user

print
print("Simple program to generate the SHA-256 Hash of the String 'Python forensics'")
print
# define a string with the desired text myString ¼ "Python forensics"

# create an object named hash which is of type sha256 hash ¼ hashlib.sha256()
# utilize the update method of the hash object to generate the # SHA 256 hash of myString
hash.update(myString)

# obtain the generated hex values of the SHA256 Hash # from the object
# by utilizing the hexdigest method

hexSHA256 ¼ hash.hexdigest()
# print out the result and utilize the upper method # to convert all the hex characters to upper case
print("SHA-256 Hash: " + hexSHA256.upper()) print
print("Processing completed")
```

This simple example demonstrates how easy it is to access and utilize the

capabilities of the Python Standard Library, and how the same code runs on different platforms and generates the correct SHA256 hash value (see Figures 2.15 and 2.16).

In Chapter 3, we will spend considerable time and focus on the application of the one-way cryptographic hash algorithms for use in digital investigation and forensics.

In the next section, we will take a deeper dive into the Python Standard Library

```
C:\Users\0\Desktop>python hashPrint.py
Simple program to generate the SHA-256 Hash of the String Python Forensics
SHA-256 Hash: 7A0BDF5725E0E032349871C8409522C0BF6971975C63F3F8041E2522148B9CF3
Processing completed
C:\Users\0\Desktop>
```

FIGURE 2.15
Windows execution of hashPrint.py.

```
chet@PythonForensics: ~/Desktop/Python Samples
chet@PythonForensics:~/Desktop/Python Samples$ python hashPrint.py

Simple program to generate the SHA-256 Hash of the String Python Forensics

SHA-256 Hash: 7A0BDF5725E0E032349871C8409522C0BF6971975C63F3F8041E2522148B9CF3

Processing completed
chet@PythonForensics:~/Desktop/Python Samples$
```

FIGURE 2.16
Ubuntu Linux execution of hashPrint.py.

WHAT IS INCLUDED IN THE STANDARD LIBRARY?

The Python Standard Library is broken down into broad categories: I will give you a brief summary description of the contents of each category with special attention on categories that I believe are unique or have forensic or digital investigation value. They are:

Built-in functions

Built-in functions, as the name implies, are always available to the Python programmer and provide fundamental capabilities that are additive to the language itself.

hex() and bin()

One of the capabilities that forensic investigators often encounter is the need to display variables in different bases; for example, from decimal (base 10), binary (base 2), and hexadecimal, or hex (base 16). These display capabilities are part of the built-in functions. The simple Python Shell session depicted in Figure 2.17 demonstrates their behavior. In this example, we set the variable a equal to the decimal

FIGURE 2.17
Python Shell session using hex() and bin().

FIGURE 2.18
Python Shell session entering hex values.

number 27 (the default is a decimal number). We then execute the function hex(a),it displays the value of a in hexadecimal form; note the use of 0x prefix to

denote hex. Utilizing the binary conversion function bin(a) renders the decimal value 27 as a binary string of 1 s and 0 s, or 11011; note the use of the 0b prefix to denote binary.

As shown in Figure 2.18, you can perform the reverse by specifying a variable as a hex number and then displaying the variable in binary or decimal form. Note that the values stored in the variable a or b shown in these examples are simply integer values, the function hex(), bin(), and str() simply render the variables in hexadecimal, binary, or decimal notation. In other words the variable does not change, just the way we view it does.

range()

Another useful built-in function is range(). We often need to create a list of items, and this built-in function can assist us in automatically creating such a list. Lists, Tuples, and Dictionaries are quite powerful constructs in Python, and we will be using each of these to solve challenge problems throughout the book. For now I am just going to show you the basics.

I have executed a few examples using the Python Shell as shown in Figure 2.19. The first one creates a list of the first 20 integers starting at zero. The second provides

FIGURE 2.19
Python Shell creating lists using the range() built-in Standard Library function.

```
C:\Windows\system32\cmd.exe

C:\Users\O\Desktop>python ipRange.py
192.168.0.0
192.168.0.1
192.168.0.2
192.168.0.3
192.168.0.4
192.168.0.5
192.168.0.6
192.168.0.7
192.168.0.8
192.168.0.9
192.168.0.10
192.168.0.11
192.168.0.12
192.168.0.13
192.168.0.14
192.168.0.15
192.168.0.16
192.168.0.17
192.168.0.18
192.168.0.19

C:\Users\O\Desktop>_
```

FIGURE 2.20
ipRange execution.

a list of integers starting at 4 and ending at 22 (notice this list stops at 22 and does not include it in the list). Finally, I create a list from 4 to 22 in steps of 3.

You will see later how using the range()function and lists can be helpful in automatically generating lists of useful values. For example, the program below generates a list that contains the first 20 host IP addresses for the class C address starting with 192.168.0. You can see the program output in Figure 2.20.

define a variable to hold a string representing the base address baseAddress ¼ “192.168.0.”
next define a list of host addresses using the range
standard library function (this will give us values of 1-19 hostAddresses ¼ range(20)
define a list that will hold the result ip strings
this starts out as a simple empty list
ipRange ¼ []
loop through the host addresses since the list hostAddresses # contains the integers from 0-19 and we can create
a loop in Python that processes each of the list elements # stored in

hostAddresses, where i is the loop counter value for i in hostAddresses:
# append the combined ip strings to the ipRange list
# because ipRange is a list object, the object has a set of # attributes and methods. We are going to invoke the append method # each time through the loop and concatenate the base address # string with the string value of the integer
ipRange.append(baseAddress+str(i))

| | | | |__ value of the host address # | | | |__ function to convert int to str # | | |__ The string “192.168.0”
| |__ The append method of the list ipRange # |__ The list object ipRange
#
# Once completed we want to print out the ip range list # here we use the print function and instruct it to print out # the ipRange list object, I wanted to print out each of the # resulting ip address on a separate line, so I looped
# through the ipRange list object one entry at a time.
for ipAddr in ipRange:

print ipAddr

Other built-in functions
The following is the complete list of built-in functions taken from the Python Standard Library documentation. You can get more information on each of the functions at the URL http://docs.python.org/2/library/functions.html

Built-in functions for Python version 2.x are given in Table 2.1. I will be using a good number of these functions in Chapters 3–11, the Cookbook sections.
Table 2.1 Python 2.7 built-in functions

abs()
all()
any()
basestring() bin()
bool()
bytearray()
callable()
chr()
classmethod() cmp()
compile()
complex()

delattr()
dict()
dir()
divmod()
enumerate() eval()
execfile()
file()
filter()
float()
format()
frozenset() getattr()
globals()
hasattr()
hash()
help()
hex()
id()
input()
int()
isinstance() issubclass() iter()
len()
list()
locals()
long()
map()
max()
memoryview() min()
next()
object()
oct()
open()
ord()
pow()
print()
property() range()
raw_input() reduce()
reload()
repr()
reversed() round()

set()
setattr()
slice()
sorted()
staticmethod() str()
sum()
super()
tuple()
type()
unichr()
unicode()
vars()
xrange()
zip()
__import__() apply()
buffer()
coerce()
intern()

Built-in constants

There are several built-in constants within Python. Constants are different from variables in that constants, as the name implies, never change—while variables can take on ever changing values.

The two most important built-in constants within Python are True and False, which are Boolean values. Take a look at Figure 2.21, I have used the Python Shell to define two variables false¼ 0 and true¼ 1 (notice these are lower case). When I use the built-in Python function type() to identify the variable type, you notice that it returns the value of int (integer) for both true and false. However, when I use type() on True and False (the built-in constants) it returns type bool (Boolean).

By the same token you can create variables a¼ True and b¼ False (as shown in Figure 2.22) and check the type() and you see that they are of type bool. Do not get too dependent on this, however, as Python is NOT a strongly typed language.Ina strongly typed language, a variable has a declared type that cannot change during the

```
C:\Python27\python.exe
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> false = 0
>>> true = 1
>>>
>>> type(false)
<type 'int'>
>>> type(true)
<type 'int'>
>>>
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
>>>
>>>
```

FIGURE 2.21
Built-in True and False constants.

```
C:\Python27\python.exe
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> a = True
>>> b = False
>>>
>>> type(a)
<type 'bool'>
>>>
>>> type(b)
<type 'bool'>
>>>
>>> a = range(100, 400, 2)
>>> type(a)
<type 'list'>
>>>
```

FIGURE 2.22
Python is not a strongly typed language.

life of the program. Python does not have this restriction; I could just as easily in the very next statement declare a¼range(100, 400, 2) and change the type of variable a from a bool into a list. To a hard core developer in a strongly typed language such as Ada, C#, Java, or Pascal, this lack of structure would have them run for the hills, however, with the proper discipline in naming objects we can control most of the chaos.

When we develop our first real application in Chapter 3, I will cover some tips on how to develop forensically sound programs using a weakly typed language.

A strongly typed language must define the variable or object along with the specific type. This type must not change once declared and the compiler itself

must enforce the typing rules, not the programmer. For example, if a variable is declared as an integer, it would be illegal to assign the value 3.14 to the variable.

Built-in types
Python has many built-in types that we can take advantage of in our forensic applications. For a detailed description of each of the standard built-in types, the most upto-date information is available at:
http://docs.python.org/2/library/stdtypes.html

The basic categories include:

Numeric Types : int, float, long and complex
Sequence Types: list, tuple, str, unicode, bytearray and buffer Set Types: set and frozenset
Mapping Types: dict or dictionary
File Objects: file
Memory: MemoryView Type

We have already seen a few of these in action during the simple examples. We will be extensively leveraging some of the more advanced types throughout our example programs including: bytearray, lists, dictionaries, unicode, sets, frozensets, and the MemoryView Type. All of these have very unique forensic applications.

Some of the bitwise operations that are included in Python are vital to many forensic and examination operations. They are quite standard, but it is worth covering to give you a flavor of what is available. Bitwise operations are exclusive to integer data types of virtually any size. For the examples below the values of x and y are the integers.

x|y : bitwise or of the variables x and y
x ^y: bitwise exclusive or of variables x and y
x&y: bitwise and of variables x and y
x << n: variable x is shifted n bits left
x >> n: variable x is shifted n bits right
x: the variable x bits are inverted

In Figure 2.23, I use the bitwise exclusive or operator ^which is used in many one-way hash and cryptography operations. We start out by setting up our

variables

```
C:\Python27\python.exe
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> x=152
>>>
>>> y=103
>>>
>>> z=x^y
>>>
>>> print z
255
>>>
>>> print hex(z)
0xff
>>>
>>> print bin(z)
0b11111111
>>>
```

FIGURE 2.23
Apply an Exclusive OR (XOR)

x and y. We set x equal to 152 and y equal to 103—both decimal numbers. We then set z equal to the exclusive or of x and y. We can then print out the decimal, hex, and binary representations of the result.

As you can see exclusive or (XOR) differs from a simple OR operation. XOR requires that each bit of the two numbers be evaluated and a one is returned if and only if, only one of the bits is set to a one. For example:

Binary 101
Binary 001
Produces:100

The least significant bit of each binary value is 1, therefore it does not meet the exclusivity test and the resulting value is 0.

Other more complex built-in types such as memory view types, byte arrays, list, unicode, and dict provide building blocks to tackle difficult data carving, search, indexing, and analytical analysis of the most complex digital evidence types available. I will be leveraging those extensively as we move into the example chapters.

Built-in exceptions
Most modern software languages support inline exception handling and Python is no different. Exception handling in most applications is important, but within

forensic and digital investigation applications it is vital. When developing these applications it is important that you can demonstrate that you have handled all the possible situations. When developing an application we all tend to test our code using “happy mode testing.” Ronda Caracappa, one of the best software quality people I have ever known, coined the phrase “happy mode testing” many years ago. This means that we test the normal flow of our programs with all the inputs, events, and third-party modules behaving exactly as we expect them to. The problem is they rarely, if ever do just that.

To facilitate exception handling, Python uses the try/except model. Here is an example: We setup our program to divide two numbers—in this case integer 27 divided by 0. If this were to happen while a program was running and we did not setup to catch such errors, the program would fault and crash. However, by using exception handling we can avoid trouble and handle the fault. In Chapter 3, our first real program, we will encounter several real-life conditions where exception handing is vital. And, with the help of Python’s built-in exception handling, we can deal with the potential pitfalls with ease and use these exceptions to log any processing difficulties.

```
x ¼ 27
y ¼ 0
try:

z ¼x/y
except:
print(“Divide by Zero”)
```

Note this is a simple example of using the try/except method to catch execution-based exceptions, in a neat and tidy manner. Other examples dealing with operating system, file handling, and network exceptions will be demonstrated during the cookbook chapters.

File and directory access

The Python Standard Library provides a rich set of file and directory access services. These services not only provide the expected ability to open, read, and write files but they also include several very unique built-in capabilities; for example, common path name manipulations. Operating systems and platforms handle directory paths differently (i.e., Windows uses the c:\users\... notation while Unix environments utilize /etc/... notation), and some operating

environments support simple ASCII file and directory naming conventions, while others support full Unicode naming conventions. All file and directory access functions need to support these differences uniformly otherwise cross-platform operations would not be possible. In addition, built into the Python Standard Library file are directory compare functions, automatic creation of temporary directories and high-level file input handling which makes dealing with file systems easier for both novice and pros.

Data compression and archiving

Instead of having to use third-party libraries to perform standard compression and archiving functions like zip and tar, these functions are built-in. This includes not only compressing and decompressing archives but also includes extracting content information for zip files. During investigation, we often encounter encrypted zip files, and Python handles these as well (provided you either have the password or you can of course apply dictionary or brute force attack methods).

File formats

Also built into the Standard Library are modules to handle special file formats; for example, comma separated value (CSV) files, the Hypertext Markup Language (HTML), the eXtensible Markup Language (XML), and even the JavaScript Object Notation (JSON) format. We will be using these modules in later chapters to carve data from Web pages and other Internet content, and to create standardized XML output files for generating reports.

Cryptographic services

We will make extensive use of the hashlib cryptographic module to solve some basic problems that face digital investigators in Chapter 3.The hashlib module not only directly supports legacy one-way cryptographic hashing such as MD5 but also directly supports modern one-way hashing algorithms such as SHA-1, SHA256, and SHA512. The ease of integration and use of these libraries along with the optimized performance provides direct access to functions that are vital to digital investigation.

Operating system services

The operating systems (OS) services provide access to core OS functions that work across platforms. In Figure 2.24 I utilize the os module to list the contents of current working directory. The steps are pretty simple:

1. Import the os module
2. We use the os modules os.getcwd() method to retrieve the current working directory path, in other words the path we are currently on
3. We store that value in the variable myCWD

```
C:\Python27\python.exe
>>> import os
>>>
>>> myCWD = os.getcwd()
>>>
>>> dirContents = os.listdir(myCWD)
>>>
>>> for names in dirContents:
...     print names
...
DLLs
Doc
include
Lib
libs
LICENSE.txt
NEWS.txt
python.exe
pythonw.exe
README.txt
Tools
w9xpopen.exe
>>>
```

FIGURE 2.24
Example using the os module from the Standard Library.

4. We then use the os.listdir() method to obtain the names of the files and directories in the current working directory
5. Instead of just printing out os.listdir(), i.e., print(os.listdir()) I decided to store the results into a new list named dirContents
6. I then use dirContents to print out each name on a separate line by processing through the list with a simple for loop

Having the files in a list enables us to process each of the files or directories based upon our needs.

Other operating system services such as stream-based io, time, logging, parsing, and platform modules all provide easy access to operating system services. Once again, on a cross-platform basis this is to easily handle Windows, Linux, Mac, and numerous legacy systems.

Standard Library summary
This has been just a quick introduction and tour of only a few of the Standard

Library capabilities built directly into Python. There are many good references and complete books dedicated to this subject for those that want to delve further. However, there are also online resources available directly at Python.org, where you can find a plethora of tutorials, code examples, and support for every Standard Library data type, module, attribute, and method. My goal here was just to give you a taste of what you can do and how easy it is to use the Standard Library.

In the next section, I will introduce you to some of the third-party libraries that we will be using and what they bring to the party.

THIRD-PARTY PACKAGES AND MODULES

As I mentioned earlier, many third-party packages and modules are available and I will introduce each during the cookbook Chapters 3–11. However, I wanted to mention a couple here to give you an idea why third-party modules are available and the kind of capabilities that they bring to digital investigation.

The natural language toolkit [NLTK]

Today when performing investigations, e-discovery, or incident response, it is important to determine what was communicated via e-mail, text message, written documents, and other correspondence. We utilize crude tools and technologies today like simple keyword and phrase searching and grep to discover and parse these communications.

Grep, which is short for global regular expression print, allows users to search text for the occurrence of a defined regular expression. Regular expressions are composed of meta characters and usually a sequence of text characters that represent a given pattern, which allow the computer program to perform advanced or simple pattern matching.

The problem is these technologies do not consider language or semantics, thus it is easy to miss certain communications or maybe misunderstand them. The NLTK modules provide an infrastructure to create real Natural Language programs and build small or even large corpus.

A corpus is a collection of material which is most often written, but can be spoken material as well. The material is digitally organized in such a way that it lends itself to the study of linguistic structures which helps to understand meaning.

This gives us the ability to search, understand, and derive meaning. We will be using NLTK in one of our Cookbook examples to help produce specific applications to assist digital investigators. For example, we can use linguistics to help determine if specific documents were likely to have been written by a specific person.

Twisted matrix [TWISTED]
In a world where everything is connected, our ability to collect and examine evidence requires both a postmortem and a live investigation mindset. This implies that we must have a reliable network library that provides an asynchronous and event-based environment that is open source and written in Python. This includes TCP, UDP, Web Services, Mail Services, Authentication, Secure Shell support, and quite a bit more.

INTEGRATED DEVELOPMENT ENVIRONMENTS

As I mentioned at the outset of this chapter, one of the keys to success would be the establishment of an IDE that would provide value, instill confidence, and create efficiencies during the development process. In our case, we also want to ensure that the quality of what we create will exceed the standards of evidence. You have several options for an IDE—some of which are completely free—and some which have a nominal cost of entry. I will reveal my choice at the end of the section, but you can choose your own path, as in many ways this is like choosing between Nike® or Adidas®andismostly definedby whatyou like andare comfortable with.

What are the options?
Actually, it may require a whole book to answer that question! Here is just a short list of the available options: IDLE, PyCharm, PyDev, WingIDE, andmDev. I will narrow my focus to just two—IDLE and WingIDE.

IDLE
IDLE specifically stands for Integrated Development Environment and the name was chosen by Guido Van Russom the creator of Python. IDLE is written in Python and has a nice set of features that can handle basic integrated development. It includes a Python Shell to experiment with commands; it highlights source code and provides cross-platform support. IDLE does support a debugger allowing users to set breakpoints and step through code (Figure 2.25). However, it was not developed as a professional development

environment, but it does support users that are developing simple applications, and students that are learning the language. Best thing is that it is completely free. Please note, I am sure that there are many Python developers

FIGURE 2.25
Python IDLE integrated development environment.

out there that use Python IDLE and will disagree with my statement about not considering it as a professional development environment, as I am sure many of you have developed quite sophisticated applications with IDLE and I accept that.

But I think for developing digital investigation applications today, there are several other alternatives that have more advanced capabilities.

WingIDE
WingIDE on the other hand is not free (you can obtain a free version if you are a student or unpaid open source developer), but does have a rich feature set that will easily support the development of sophisticated Python Applications. WingIDE comes in three flavors based on your intended use:

1. Free for students and unpaid open source developers
2. WingIDE Personal—As the name implies, limits some features
3. WingIDE Professional—For those wanting all the features and support, for example, this version contains unit testing, direct interface with revision control systems, advanced debugging and breakpoint settings, and pyLint integration.

pyLint is a Python source code analyzer that helps to identify poor coding practices, points out potential bugs, and warns you about the likelihood of failure. It will also assign an overall grade for your code to help you to keep improving.

You can see the layout I have chosen for WingIDE in Figure 2.26. The IDE is configurable and resizable to allow you to customize the GUI. I have placed labels in each of the major sections in order to cover them here.

Section A : In this section, we find the current local and global variables associated with the current running program, since I have the program momentarily stopped at a breakpoint. As you can see, the text box at the top indicates that module ipRange.py is currently processing line 10. In the window it displays the variable baseAddress currently containing the string “192.168.0.”

Section B : Displays the program output so far. The initial print statement has successfully printed the string “Generating ip Range.” At the bottom of that window you can see several other options; the currently selected option is “Debug I/O.” Another quite useful option is the “Python Shell.” If this option is selected you can execute, learn, or experiment with any of the Python functions or even write a sequence of statements to try before you code (see Figure 2.27).

Section C : This window contains the project information. Due to the fact this is a very simple project only containing a single file, ipRange.py, that is all that is displayed in that window. As we build more complex applications this window

will help you keep track of all the components of your program.

Section D : This window contains the source code of the current program, along with debug settings. Notice that line 10 contains a dot where the breakpoint was set and it is highlighted because the program reached that point and the breakpoint stopped execution.

FIGURE 2.26
Snapshot of WingIDE 4.1 Personal.

FIGURE 2.27
WingIDE Python Shell display.

Now that you have the general idea, let us actually step through this simple program with WingIDE and watch the changes happen. In Figure 2.28 we step over (I used the [F6] key to do this, you could also select the option under the debug menu) the code below and stop before executing the ipRange initialization.

baseAddress ¼ “192.168.0.”
minorAddress ¼ range(20)
ipRange ¼ []

FIGURE 2.28
WingIDE in action.

At this point we can examine the variable section top left (in Figure 2.28) and we see that the variable baseAddress is equal to the string “192.168.0” and the variable hostAddress is a list with length 20 and has values 0-19 as expected. We name this variable hostAddress because in typical Class C address, the last

dotted decimal digit typically identifies the host. We consider naming conventions for variables, functions, methods, and attributes as we get into the Cookbook chapters, but I am sure you can already see the pattern.

Next, I am going to step over the initialization of the ipRange (again using [F6]) and also step into the “for loop” stopping before we execute the first ipRange.append method. As you can see in Figure 2.29, the new variables i which equals 0, and is the first hostAddress, and ipRange, which is currently an empty list, have now appeared.

Now I am going to step through each of the for loop items, 20 iterations in this case and stop on the first print statement of the second loop. You notice in Figure 2.30 in the variable section, I have expanded the contents of the ipRange list and you see the completed ipRange strings that now contain the added host addresses. All that is left is to print out the ipRange array line by line.

This should give you a good overview of the capabilities of WingIDE and how to use it to set breakpoints, step through the code, and examine variables. I will be using this IDE throughout the rest of the book introducing advanced features as we go.

FIGURE 2.29
Using WingIDE to step through code.

One final feature that I use a great deal when writing code is auto-completion. In Figure 2.31, I want to see what other methods and attributes are associated with the list ipRange. We have seen what the append method does, but you might be wondering what else we can do. By simply adding a line of code, ipRange, and entering a dot (a period), a list box automatically appears and displays all the attributes and methods that are available for ipRange. Note that since ipRange is a list, these are actually the attributes and methods that can be generally applied to any list. The dropdown you see is only a partial list of methods and attributes. As you can see, I have scrolled down to the sort method which would allow me to sort the list. The parameters that I pass to the sort method will determine how the list will actually be sorted.

Up to this point we have focused on developing and debugging on a Windows platform. In the next section, we will walk through an Ubuntu Linux installation

and quick session.

Python running on Ubuntu Linux

There are many investigation and forensic advantages to Linux-based tools. You can of course achieve greater performance, safely mount a wide range of media and image types, avoid the cost of the operating system, and have potentially greater flexibility. So let us setup Python running on Ubuntu. It really is easier than you might think.

FIGURE 2.30

WingIDE examination of the completed list.

FIGURE 2.31
WingIDE auto complete feature.

FIGURE 2.32

Ubuntu download Web page 12.04 LTS.

As of this writing, I would make the following strong recommendation on Ubuntu installation and setup. In Figure 2.32 you see a screenshot from the Ubuntu download page (http://www.ubuntu.com/download/desktop). Version 12.x LTS of Ubuntu is your best bet. This version is available in either 32 or 64 bit and is verified to operate on a wide range of standard desktop PCs. I am choosing version 12.x LTS because this is the long-term support version of the OS. Ubuntu is promising to provide support and security updates for this version through April 2017, giving you a stable well-tested platform for digital investigation and forensics.

As far as installing and running Python on the Ubuntu operating system, version 2.7.3 is already installed as part of the OS, thus installation of the base product is done for you. If you are not sure about your specific installation, simply launch a terminal window (as shown in Figure 2.33), and type Python at the prompt. If it is installed properly you will see a similar message as the one shown.

If this does not produce the desired results, you should reinstall Ubuntu 12.x LTS and this should repair the problem. (Make sure you backup your data before attempting any reinstall.)

If you have not used Linux in a while, or this is your first time installing Ubuntu, you will find that the days of struggling to install a Linux OS are over. As the saying goes “it is as easy as pie.” Once you have successfully installed the OS, the process of adding new capabilities is straightforward. For example, if you intend to install an IDE like WingIDE or IDLE in the Ubuntu environment, you can obtain the installers from the Ubuntu Software Center. The Ubuntu software center can be contacted by clicking on the taskbar item show in Figure 2.34, and searching for your desired application.

```
chet@PythonForensics: ~
chet@PythonForensics:~$
chet@PythonForensics:~$ python
Python 2.7.3 (default, Aug  1 2012, 05:16:07)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

FIGURE 2.33
Ubuntu terminal window Python command.

FIGURE 2.34
Ubuntu software center.

FIGURE 2.35
WingIDE running on Ubuntu 12.04 LTS.

As you see, I have searched for Python and the Software Center displays all the relevant applications and services for me. You notice the first item that I have circled for you is the actual Python environment. The Ubuntu Software Center shows that this application is already installed (notice the checkmark in front of the icon). Also, you see that two different versions of the IDLE IDE are also listed one supporting version 2.7 and the other supporting version 3.2.

I went ahead and installed the WingIDE environment for Ubuntu as this is the environment I have chosen. You can see the screenshot in Figure 2.35 where WingIDE is running under Ubuntu 12.04 LTS, with the same feature set and operation as the Windows version that you saw earlier in this chapter.

PYTHON ON MOBILE DEVICES

Progress is currently being made to port versions of Python to smart mobile

devices (iOS, Windows 8 Phones, and of course Android). The apps that have been created have some limited functionality, but they are worth considering for experimentation or learning. I will take a look at a couple of these in the next section.

iOS Python app

Python for iOS is the version that I find most stable and supported for iPad. In Figure 2.36, you can see an image of the Python 2.7 Shell running on iOS, just

FIGURE 2.36
Python Shell running on iOS.

printing out the similar Hello World message. The Shell and editor have some great features that allow you to experiment with the Python language syntax and modules. The capability is actually quite a bit better than this simple example. I imported one of the earlier examples that performs a SHA256 Hash of a string. At the top of the short program I added a couple of new methods, one contained in the Standard Library module sys and one in the Standard Library module

platform. I imported the two modules and wrote the following code.

```
import sys
import platform
print("Platform: "+sys.platform)
print("Machine: "+platform.machine())
```

You can see the results of these new methods and the program in Figure 2.37. You notice that Platform is defined as Darwin and the Machine is identified as an iPad version 1. Correct in both cases. Darwin is the platform name specified by Apple for Mac OS X and of course the machine is my old and still faithful iPad 1.

For those interested in checking out this iOS application from the Apple App Store, Figure 2.38 is a screenshot from the Store with more details about Python for iOS.

FIGURE 2.37

iOS implementation of HashPrint.

FIGURE 2.38
Apple App Store page regarding Python for iOS.

Windows 8 phone

Finally, we can take a look at the Microsoft Windows 8 OS on a Nokia Lumia Phone. Yes—Python on a Phone. I tried several applications from the Microsoft store before I found one that worked pretty well. PyConsole (short for Python Console), shown in the launcher menu in Figure 2.39, ran the sample hashPrint.py program unchanged.

As we progress through the book, we will come back to these devices and others to see how well the more advanced applications work on each of them.
When the app launches it brings up a simple Python Shell as shown in Figure 2.40. I typed in a simple Hello Mobile World message, and you can see that I ran the script. The result appears in the display windows at the bottom of the screen. Turning to the simple SHA256 hashing example shown in Figure 2.41, we see the code and execution of the program producing the same results as the other

platforms.
If you are interested in the Python Console Windows 8 Phone application, Figure 2.42 depicts a screenshot from the Windows Store.

FIGURE 2.39
Windows 8 Phone screenshot of PyConsole launching

FIGURE 2.40
Python Console “Hello World” on a Windows 8 Phone.

FIGURE 2.41
Windows 8 Phone HashPrint application execution.

FIGURE 2.42 Python Console Windows App Store page.

Summary Questions 51

A VIRTUAL MACHINE

Today, we have virtual machines for everything from complete server infrastructures to specialized applications, standardized development environments, databases, client services, desktops, and more. For those who want a really quick start, I have created a Ubuntu Python Environment that will be available in conjunction with the publication of this book. The environment includes a standard Ubuntu install, all the packages, modules, and Cookbook applications from Chapters 3 to 11. I have also included any test data so you can start experimenting with the simplest or most advanced digital investigation or forensic applications found within. You can access the virtual machine by going

to www.python-forensics.org.

CHAPTER REVIEW

In this chapter we took a very broad, and in some cases deep, look at setting up a Python environment in the Microsoft Windows and Linux environment. We reviewed where to download the latest Python environment from the official source and walked through the steps of installing a Windows Python environment. We also made specific recommendation on the Ubuntu version of choice that include Python version 2.x already installed as part of the OS. We looked at the key differences between Python 2.x and 3.x. We also pulled back the covers a bit on the Python Standard Library and dialed in on some of the key built-in functions, data types, and modules. We even developed a few example applications that exercised the Standard Library and showed how a program written in Python could run without modification on Windows, Linux, iOS, or on a Windows 8 phone. MacOS is also supported and Python 2.x comes with the standard install of the latest versions.

We examined two specific third-party packages that address Natural Language and network applications, respectively. We also examined some of the capabilities that we will need included in an IDE, and took a step-by-step look at the features of WingIDE and briefly covered the IDLE IDE. Finally, we experimented with the Python Shell on all the computing platforms mentioned here.

SUMMARY QUESTIONS

1. What do you think would be the best Python development environment based on the following circumstances?
a. A student looking to experiment with Python forensics
b. A developer that plans to build and distribute Python investigative solutions c. A laboratory that processes actual cases, but needs specialized tools and

capabilities that do not exist within standard forensic tools

2. What are some of the key features you would be looking for in an IDE if you are creating and utilizing Python forensic applications on actual cases or during real investigations?

3. What version of Ubuntu Linux would you choose for your platform of choice and why?
4. How might you utilize mobile platforms like iOS, Android, or Windows 8 devices for investigative purposes when a fully functional Python environment is available for them?
5. What are some of the key considerations for choosing Python 2.x vs. 3.x today?
6. What third-party modules or package would assist in Natural Language Processing?
7. What third-party modules or packages would assist in creating asynchronous network applications?

LOOKING AHEAD

We are now going to transition into the Cookbook chapters of the book, where in each chapter we tackle a specific forensic or digital investigation challenge. We will define the challenge, specify requirements, design an approach, code a solution, and finally test and validate the solution.

As we go forward, I will be slowly introducing new language constructs, packages and modules, debugging methodologies, and good coding practices in each chapter. So, I hope by this time you have setup your Python IDE and are ready to get started.

Additional Resources

Python Programming Language—Official Website, Python.org.
http://www.python.org. The Python Standard Library.
http://docs.python.org/2/library/.
The Natural Language Toolkit. nltk.org.
Twisted Network Programming for Python. http://twistedmatrix.com.

CHAPTER

Our First Python Forensics App 3

CHAPTER CONTENTS

Introduction .....

54
Naming Conventions and Other Considerations
..... 55
Constants 55
Local Variable Name 55
Global Variable Name 55
Functions Name 55
Object Name 55
Module 55
Class Names 56
Our First Application “One-Way File System Hashing”
..... 56
Background 56
One-way Hashing algorithms’ Basic Characteristics 56
Popular Cryptographic Hash Algorithms? 57
What are the Tradeoffs Between One-Way Hashing Algorithms? 57
What are the Best-use Cases for One-Way Hashing Algorithms in Forensics? ...
57
Fundamental Requirements 58
Design Considerations 59
Program Structure 61
Main Function 62
ParseCommandLine 62
WalkPath Function 63
HashFile Function 63
CSVWriter (Class) 63
Logger 63
Writing the Code 63
Code Walk-Through
..... 64
Examining Main—Code Walk-Through 64
ParseCommandLine() 66
ValiditingDirectoryWritable 69
WalkPath 69
HashFile 71
CSVWriter 74
Full Code Listing Pfish.py 75
Full Code Listing _Pfish.py 76
Results Presentation .....

83 Chapter Review ..... 86 Summary Questions ..... 89 Looking Ahead ..... 89 Additional Resources 89

© 2014 Elsevier Inc. All rights reserved.

INTRODUCTION

In 1998, I authored a paper entitled “Using SmartCards and Digital Signatures to Preserve Electronic Evidence” (Hosmer, 1998). The purpose of the paper was to advance the early work of Gene Kim, creator of the original Tripwire technology (Kim, 1993) as a graduate student at Purdue. I was interested in advancing the model of using one-way hashing technologies to protect digital evidence, and specifically I was applying the use of digital signatures bound to a SmartCard that provided twofactor authenticity of the signing (Figure 3.1).

Years later I added trusted timestamps to the equation adding provenance, or proof of the exact “when” of the signing.

Two-factor authentication combines a secure physical device such as a SmartCard with a password that unlocks the capability of the card’s. This yields “something held” and “something known.” In order to perform applications like signing, you must be in possession of the SmartCard and you must know the pin or password that unlocks the cards function.

Thus, my interest in applying one-way hashing methods, digital signature algo

Thus, my interest in applying one-way hashing methods, digital signature algo
year journey ... so far. The application of these technologies to evidence preservation, evidence identification, authentication, access control decisions and network protocols continues today. Thus I want to make sure that you have a firm understanding of the underlying technologies and the many applications for digital investigation, and of course the use of Python forensics.

FIGURE 3.1
Cryptographic SmartCard.

Naming Conventions and Other Considerations 55

Before I dive right in and start writing code, as promised I want to set up some ground rules for using the Python programming language in forensic applications.

NAMING CONVENTIONS AND OTHER CONSIDERATIONS

During the development of Python forensics applications, I will define the rules and naming conventions that are being used throughout the cookbook chapters in the book. Part of this is to compensate for Python's lack of the enforcement of strongly typed variables and true constants. More importantly it is to define a style that will make the programs more readable, and easier to follow, understand, and modify or enhance.

Therefore, here are the naming conventions I will be using.

Constants

Rule: Uppercase with underscore separation Example: HIGH_TEMPERATURE

Local variable name

Rule: Lowercase with bumpy caps (underscores are optional) Example: currentTemperature

Global variable name

Rule: Prefix gl lowercase with bumpy caps (underscores are optional) Note: Globals should be contained to a single module Example: gl_maximumRecordedTemperature

Functions name

Rule: Uppercase with bumpy caps (underscores optional) with active voice Example: ConvertFarenheitToCentigrade(...)

Object name

Rule: Prefix ob_ lowercase with bumpy caps Example: ob_myTempRecorder
Module
Rule: An underscore followed by lowercase with bumpy caps Example: _tempRecorder

Class names
Rule: Prefix class_ then bumpy caps and keep brief
Example: class_TempSystem
You will see many of these naming conventions in action during this chapter.

OUR FIRST APPLICATION "ONE-WAY FILE SYSTEM HASHING"
The objective for our first Python Forensic Application is as follows:

1. Build a useful application and tool for forensic investigators.
2. Develop several modules along the way that are reusable throughout the book and for future applications.
3. Develop a solid methodology for building Python forensic applications.
4. Begin to introduce more advanced features of the language.

Background
Before we can build an application that performs one-way file system hashing I need to better define one-way hashing. Many of you reading this are probably saying, "I already know what a one-way hashing is, let's move on." However, this is such an important underpinning for computer forensics it is worthy of a good definition, possibly even a better one that you currently have.

One-way hashing algorithms' basic characteristics
1. The one-way hashing algorithm takes a stream of binary data as input; this could be a password, a file, an image of a hard drive, an image of a solid state drive, a network packet, 1's and 0's from a digital recording, or basically any continuous digital input.
2. The algorithm produces a message digest which is a compact representation of the binary data that was received as input.
3. It is infeasible to determine the binary input that generated the digest with only the digest. In other words, it is not possible to reverse the process using the digest to recover the stream of binary data that created it.
4. It is infeasible to create a new binary input that will generate a given message digest.
5. Changing a single bit of the binary input data will generate a unique message

digest.
6. Finally, it is infeasible to find two unique arbitrary streams of binary data that produce the same digest.

Table 3.1 Popular One-Way Hashing Algorithms

Length Algorithm Creator (Bits) Related standard

MD5 Ronald Rivest 128
SHA-1 NSA and published by NIST 160
SHA-2 NSA and published by NIST 224
256
384
512
RIPEMD-160 Hans Dobbertin 160

SHA-3 Guido Bertoni, Joan Daemen, 224, 256, Michae¨l Peeters, and Gilles Van
384, 512 Assche

Popular cryptographic hash algorithms? There are a number of algorithms that produce message digests. Table 3.1 provides background on some of the most popular algorithms.

RFC 1321
FIPS Pub 180 FIPS Pub 180-2 FIPS Pub 180-3 FIPS PUB 180-4

Open Academic Community
FIPS-180-5

What are the tradeoffs between one-way hashing algorithms? The MD5 algorithm is still in use today, and for many applications the speed, convenience, and interoperability have made it the algorithm of choice. Due to attacks on the MD5 algorithm and the increased likelihood of collisions, many organizations are moving to SHA-2 (256 and 512 bits are the most popular sizes). Many organizations have opted to skip SHA-1 as it suffers from some of the same weaknesses as MD5.

Considerations for moving to SHA-3 are still in the future, and it will be a couple of years before broader adoption is in play. SHA-3 is completely different and was designed to be easier to implement in hardware to improve performance

(speed and power consumption) for use in embedded or handheld devices. We will see how quickly the handheld devices' manufacturers adopt this newly established standard.

What are the best-use cases for one-way hashing algorithms in forensics? Evidence preservation: When digital data are collected (for example, when imaging a mechanical or solid state drive), the entire contents—in other words every bit collected—are combined to create a unique one-way hashing value. Once completed the recalculation of the one-way hashing can be accomplished. If the new calculation matches the original, this can prove that the evidence has not been modified. This assumes of course that the original calculated hash value has been safeguarded against tampering since there is no held secret and the algorithms are available. Anyone could recalculate a hash, therefore the chain of custody of digital evidence, including the generated hash, must be maintained.

Search : One-way hashing values have been traditionally utilized to perform searches of known file objects. For example, if law enforcement has a collection of confirmed child-pornography files, the hashes could be calculated for each file. Then any suspect system could be scanned for the presence of this contraband by calculating the hash values of each file and comparing the resulting hashes to the known list of contraband hash values (those resulting from the child-pornography collection). If matches are found, then the files on the suspect system matching the hash values would be examined further.

Black Listing : Like the search example, it is possible to create a list of known bad hash files. These could represent contraband as with CP example, they could match known malicious code or cyber weapon files or the hashes of classified or proprietary documents. The discovery of hashes matching any of these Black Listed items would provide investigators with key evidence.

White Listing : By creating a list of known good or benign hashes (operating system or application executables, vendor supplied dynamic link libraries or known trustworthy application download files), investigators can use the lists to filter out files that they do not have to examine, because they were previously determined as a good file. Using this methodology you can dramatically reduce the number of files that require examination and then focus your attention on files that are not in the known good hash list.

Change detection : One popular defense against malicious changes to websites,

routers, firewall configuration, and even operating system installations is to hash a “known good” installation or configuration. Then periodically you can re-scan the installation or configuration to ensure no files have changed. In addition, you must of course make sure no files have been added or deleted from the “known good” set.

Fundamental requirements
Now that we have a better understanding of one-way hashing and its uses, what are the fundamental requirements of our one-way file system hash application?

When defining requirements for any program or application I want to define them as succinctly as possible, and with little jargon, so anyone familiar with the domain could understand them—even if they are not software developers. Also, each requirement should have an identifier such that could be traced from definition, through design, development, and validation. I like to give the designers and developers room to innovate, thus I try to focus on WHAT not HOW during requirements definition (Table 3.2).

Table 3.2 Basic Requirements Requirement Requirement number name
000 Overview
001 Portability
002 Key functions
003 Key results
004 Algorithm selection

005 Error handling Short description

The basic capability we are looking for is a forensic application that walks the file system at a defined starting point (for example, c:\ or /etc) and then generates a one-way hashing value for every file encountered
The application shall support Windows and Linux operating systems. As a general guideline, validation will be performed on Windows 7, Windows 8, and Ubuntu 12.04 LTS environments In addition to the one-way hashing generation, the application shall collect system metadata associated with each file that is hashed. For example, file attributes, file name, and file path at a minimum The application shall provide the results in a standard output file format that offers flexibility The application shall provide a wide diversity when specifying the one-way hashing algorithm(s) to be used
The application must support error handling and logging of all operations

performed. This will include a textual description and a timestamp

Design considerations Now that I have defined the basic requirements for the application I need to factor in the design considerations. First, I would like to leverage or utilize as many of the built-in functions of the Python Standard Library as possible. Taking stock of the core capabilities, I like to map the requirements definition to Modules and Functions that I intend to use. This will then expose any new modules either from third party modules or new modules that need to be developed (Table 3.3).

One of the important steps as a designer or at least one of the fun parts is to name the program. I have decided to name this first program p-fish short for Pythonfile system hashing.

Next, based on this review of Standard Library functions I must define what modules will be used in our first application:

1. argparse for user input
2. os for file system manipulation
3. hashlib for one-way hashing
4. csv for result output (other optional outputs could be added later)
5. logging for event and error logging
6. Along with useful miscellaneous modules like time, sys, and stat

Table 3.3 Standard Library Mapping Requirement Design considerations Library selection

User input (000, 003, 004)

Walk the file system
(000, 001)

Meta data collection (003)

File hashing (000)
Each of these requirements needs input from the user to accomplish the task. For example, 000 requires the user to specify the starting directory path. 003 requires that the user specify a suitable output format. 004 requires us to allow the user to specify the hash algorithm. Details of the exception handling or default settings need to be defined (if allowed)

This capability requires the program to traverse the directory structure starting at a specific starting point. Also, this must work on both Windows and Linux platforms
This requires us to collect the directory path, filename, owner, modified/access/created times, permissions, and attributes such as read only, hidden, system or archive
I must provide flexibility in the Hashing algorithms that the users could select. I have decided to support the most popular
algorithms such as MD5 and several variations of SHA

Result output (003)
To meet this requirement I must be able to structure the program output to support a format that provides flexibility
For this first program I have decided to use the command line parameters to obtain input from the user. Based on this design decision I can leverage the argparse Python Standard Library module

The OS Module from the
Standard Library provides key methods that provide the ability to walk the file system, OS also provides abstraction which will provide cross platform compatibility. Finally, this module contains cross platform
capabilities that provide access to metadata associated with files

The Standard Library module hashlib provides the ability to generate one-way hashing values. The library supports common hash algorithms such as “md5,” “sha1,” “sha224,” “sha256,” “sha384,” “sha512.” This should provide a sufficient set of selection for the user The Standard Library offers multiple options that I could leverage. For example, the csv module provides the ability to create comma separated value output, whereas the json module (Java Object Notation) provides encoder and decoders for JSON objects and finally the XML module could be leveraged to create XML output

Continued

Table 3.3 Standard Library Mapping—cont’d Requirement Design considerations

Logging and
error handling

I must expect that errors will occur during our walk of the file system, for example I might not have access to certain files, or certain files may be orphaned, or certain files maybe locked by the operating system or other applications. I need to handle these error conditions and log any notable events. For example, I should log information about the investigator, location, date and time, and information that pertains to the system that are walked

Program structure Next, I need to define the structure of our program, in other words how I intend to put the pieces together. This is critical, especially if our goal is to reuse components of this program in future applications. One way to compose the components is with a couple simple diagrams as shown in Figures 3.2 and 3.3.

Library selection

The Python Standard Library includes a logging facility which I can leverage to report any events or errors that occur during processing

user

p-fish Report

Program Arguments
p-fish

Event and Error Log

p-fish context diagram FIGURE 3.2
Context diagram: Python-file system hashing (p-fish).

Program Arguments

rootPath
hashType reportName ReportName

Generate Report

hashType P-fish Report

HashFile Main resultRecords rootPath fileName
WalkPath eventValue
eventValue
logName LogEvents

Event
and
Error Log

p-fish internal structure

FIGURE 3.3 p-fish internal structure.

The context diagram is very straightforward and simply depicts the major inputs and outputs of the proposed program. A user specifies the program arguments, p-fish takes those inputs and processes (hashes, extracts metadata, etc.) the file system produces a report and any notable events or errors to the “p-fish report” and the “p-fish event and error log” files respectively.

Turning to the internal structure I have broken the program down into five major components. The Main program, ParseCommandLine function, WalkPath function, HashFile functions, CSVWriter class and logger (note logger is actually the Python logger module), that is utilized by the major functions of pfish. I briefly describe the operation of each below and during the code walk through section a more detailed line by line explanation of how each function operates is provided.

Main function
The purpose of the Main function is to control the overall flow of this program. For example, within Main I set up the Python logger, I display startup and

completion messages, and keep track of the time. In addition, Main invokes the command line parser and then launches the WalkPathfunction. Once WalkPathcompletes Main will log the completion and display termination messages to the user and the log.

ParseCommandLine

In order to provide smooth operation of p-fish, I leverage parseCommandLine to not only parse but also validate the user input. Once completed, information that is germane to program functions such as WalkPath, HashFile, and CSVWrite is available from parser-generated values. For example, since the hashType is specified by the user, this value must be available to HashFile. Likewise the CSVWriter needs the path where the resulting pfish report will be written, and WalkPath requires the starting or rootPath to start the walk.

WalkPath function

The WalkPath function must start at the root of the directory tree or path and traverse every directory and file. For each valid file encountered it will call the HashFile function to perform the one-way hashing operations. Once all the files have been processed WalkPath will return control back to Main with the number of files successfully processed.

HashFile function

The HashFile function will open, read, hash, and obtain metadata regarding the file in question. For each file, a row of data will be sent to the CSVWriterto be included in the p-fish report. Once the file has been processed, HashFile will return control back to WalkPath in order to fetch the next file.

CSVWriter (class)

In order to provide an introduction to class and object usage I decided to create CSVWriter as a class instead of a simple function. You will see more of this in upcoming cookbook chapters but CSVWriter sets up nicely for a class/object demonstration. The csv module within the Python Standard Library requires that the “writer” be initialized. For example, I want the resulting csv file to have a header row made up of a static set of columns. Then subsequent calls to writer will contain data that fills in each row. Finally, once the program has processed all the files the resulting csv report must be closed. Note that as I walk through the program code you may wonder why I did not leverage classes and objects more for this program. I certainly could have, but felt for the first application I would create a more functionoriented example.

Logger

The built-in Standard Library logger provides us with the ability to write messages to a log file associated with p-fish. The program can write information messages, warning messages, and error messages. Since this is intended to be a forensic application, logging operations of the program is vital. You can expand the program to log additional events in the code, they can be added to any of the _pfish functions.

Writing the code

I decided to create two files, mainly to show you how to create your own Python module and also to give you some background on how to separate capabilities. For this first simple application, I created (1) pfish.py and (2) _pfish.py. As you may recall, all modules that are created begin with an underscore and since _pfish.py contains all the support functions for pfish I simply named it _pfish.py. If you would like to split out the modules to better separate the functions you could create separate modules for the HashFile function, the WalkPath function, etc. This is a decision that is typically based on how tightly or loosely coupled the functions are, or better stated, whether you wish to reuse individual functions later that need to standalone. If that is the case, then you should separate them out.

In Figure 3.4 you can see my IDE setup for the project pfish. You notice the project section to the far upper right that specifies the files associated with the project. I also have both files open—you can see the two tabs far left about half way down

FIGURE 3.4
p-fish WingIDE setup.

where I can view the source code in each of the files. As you would expect in the upper left quadrant, you can see the program is running and the variables are available for inspection. Finally, in the upper center portion of the screen you can see the current display messages from the program reporting that the command line was processed successfully and the welcome message for pfish.

CODE WALK-THROUGH

I will be inserting dialog as I discuss each code section. The code walk-through will give you an in-depth look at all the code associated with the program. I will be first walking through each of the key functions and then will provide you with a complete listing of both files.

Examining main—code walk-through

The embedded commentary is represented in italics, while the code itself is represented in a fixed-sized font.

```
# p-fish : Python File System Hash Program
# Author: C. Hosmer
```

July 2013
Version 1.0
#

The main program code is quite straightforward. At the top as you would expect, you see the import statements that make the Python Standard Library modules available for our use. What you have not seen before is the import statement referencing our own module in this case _pfish. Since I will be calling functions from Main that exist in our module, the module must import our own module _pfish.

import logging import time
import sys
import _pfish # Python Library logging functions
# Python Library time manipulation functions # Python Library system specific parameters # _pfish Support Function Module

if __name__ ¼¼'__main__': PFISH_VERSION ¼'1.0' # Turn on Logging

Next, you can see my initialization of the Python logging system. In this example I have hard-wired the log to be stored in the file aptly named pFishLog.log. I set the logging level to DEGUG and specified that I wanted the Time and Date recorded for each log event. By setting the level to DEBUG (which is the lowest level) this will ensure all messages sent to the logger will be visible.

logging.basicConfig(filename¼'pFishLog.log',level¼logging. DEBUG,format¼'% (asctime)s %(message)s')

Next, I pass control to process the command line arguments by calling the _pfish. ParseCommandLine() function. I must prefix the function with _pfish, since the function exists in the _pfish module. If the parse is successful, the function will return here, if not, it will post a message to the user and exit the program. I will take a deeper look at the operation of ParseCommandLine() in the next section.

Process the Command Line Arguments _pfish.ParseCommandLine()

I need to record the current starting time of the application in order to calculate elapsed time for processing. I use the Standard Library function time.time() to

acquire the time elapsed in seconds since the epoch. Note, forensically this is the time of the system we are running on, therefore if the time is a critical element in your investigation you should sync your system clock accordingly.

```
# Record the Starting Time
startTime ¼ time.time()
```

Next the program posts a message to the log reporting the start of the scan and display this on the user screen only if the verbose option was selected on the command line (more about this when I examine the ParseCommandLine function). Notice that I used a CONSTANT to hold the version number instead of just embedding a magic number. Now we can just modify the CONSTANT in the future. Then anywhere PFISH_VERSIONis used it will display the proper version number. I also logged the system platform and version in case there is a question in the future about the system that was used to process these data. This would be a great place to add information about the organization, investigator name, case number, and other information that is relevant to the case.

```
# Post the Start Scan Message to the Log
logging.info('Welcome to p-fish version 1.0 ... New Scan Started')
_pfish.DisplayMessage('Welcome to p-fish ... version 1.0')
```

Note, since I created a constant PFISH_VERSION, we could use that to make the source code easier to maintain. That would look something like:
_pfish.DisplayMessage(‘Welcome to p-fish ... ‘þ PFISH_VERSION)

```
# Record some information regarding the system
logging.info('System:'+ sys.platform)
logging.info('Version:'+ sys.version)
```

Now the main program launches the WalkPath function within the _pfish module that will traverse the directory structure starting at the predefined root path. This function returns the number of files that were successfully processed by WalkPath and HashFile. As you can see I use this value along with the ending time to finalize the log entries. By subtracting the startTime from the endTime I can determine the number of seconds it took to perform the file system hashing operations. You could convert the seconds into days, hours, minutes, and seconds of course.

Traverse the file system directories and hash the files filesProcessed ¼ _pfish.WalkPath()

Record the end time and calculate the duration
endTime ¼ time.time()
duration ¼ endTime - startTime

logging.info('Files Processed:'+ str(filesProcessed)) logging.info('Elapsed Time:'+ str(duration) +'seconds') logging.info('Program Terminated Normally') _pfish.DisplayMessage("Program End)"
ParseCommandLine()
In the design section I made a couple of decisions that drove the development:

I decided that this first application would be a command line program. 1.
2. I decided to provide several options to the user to manipulate the behavior of the program. This has driven the design and implementation of the command line options.

Based on this I provided the following command line options to the program.
Option Description Notes

-v Verbose, if this option is specified then any calls to the DisplayMessage() function will be displayed to the standard output device, otherwise the program will run silently
--MD5 Hash type selection, the user must
--SHA256 specify the one-way hashing
--SHA512 algorithm that would be utilized
-d rootPath, this allows the user to specify the starting or root path for the walk
-r reportPath, this allows the user to specify the directory where the resulting .csv file will be written The selection is mutually exclusive and at least one must be selected or the program will abort

The directory must exist and must be readable or the program will abort
The directory must exist and must be writable or the program will abort

Even though at first some of these requirements might seem difficult, the argparse Standard Library provides great flexibility in handling them. This allows us to catch any possible user errors prior to program execution and also provides us with a way to report problems to the user to handle the exceptions.

def ParseCommandLine():

The majority of the process of using argparse is knowing how to setup the parser. If you set the parser up correctly it will do all the hard work for you. I start by creating a new parser named “parser” and simply give it a description. Next, I add a new argument in this case –v or verbose. The option is –v and the resulting variable that is used is verbose. The help message associated with the argument is used by the help system to inform the user on how to use pfish. The –h option is built-in and requires no definition.

parser¼ argparse.ArgumentParser('Python file system hashing .. p-fish')
parser.add_argument('-v','--verbose'help¼'allows progress messages to be displayed', action¼'store_true')

The next section defines a mutually exclusive group of arguments for selecting the specific hash type the user would like to generate. If you wanted to add in another option, for example sha384, you would simply add another argument to the group and follow the same format. Since I specified under the add_mutually_exclusive_group the option required¼ True, argparse will make sure that the user has only specified one argument and at least one.

setup a group where the selection is mutually exclusive # # and required.
group¼ parser.add_mutually_exclusive_group(required¼True)

group.add_argument('--md5',help¼'specifies MD5 algorithm',
action¼'store_true')
group.add_argument('--sha256', algorithm', action¼'store_true') help ¼'specifies SHA256

group.add_argument('--sha512', algorithm', action¼'store_true') help ¼'specifies SHA512

Next I need to specify the starting point of our walk, and where the report should be created. This works the same as the previous setup, except I have added the type option. This requires argparse to validate the type I have specified. In the case of the –d option, I want to make sure that the rootPath exists and is readable. For the reportPath, it must exist and be writable. Since argparse does not have builtin functions to validate a directory, I created the functions ValidateDirectory() and ValidateDirectoryWritable(). They are almost identical

and they use Standard Library operating system functions to validate the directories as defined.

```
parser.add_argument( '-d','--rootPath', type¼
ValidateDirectory, required¼True, help¼"specify the root path for hashing)"

parser.add_argument( '-r','--reportPath', type¼
ValidateDirectoryWritable, required¼True, help¼"specify the path for reports
and logs will be written)"

# create a global object to hold the validated arguments, # # these will be
available then to all the Functions within # the _pfish.py module

global gl_args
global gl_hashType
```

Now the parser can be invoked. I want to store the resulting arguments (once validated) in a global variable so they can be accessible by the functions within the _pfish module. This would be a great opportunity to create a class to handle this which would avoid the use of the global variables. This is done in Chapter 4.

```
gl_args ¼ parser.parse_args()
```

If the parser was successful (in other words argparsevalidated the command line parameters), I want to determine which hashing algorithm the user selected. I do that by examining each value associated with the hash types. If the user selected sha256 for example, the gl_args.sha256 would be True and md5 and sha512 would be false. Therefore, by using a simple if/elif language routine I can determine which was selected.

```
if gl_args.md5:
gl_hashType ¼'MD5'
elif gl_args.sha256:
gl_hashType ¼'SHA256'
elif gl_args.sha512:
gl_hashType ¼'SHA512'
else:
gl_hashType ¼ "Unknown"
logging.error('Unknown Hash Type Specified')
```

DisplayMessage("Command line processed: Successfully)" return

ValiditingDirectoryWritable
As mentioned above, I needed to create functions to validate the directories provided by the users for both the report and starting or root path of the Walk. I accomplish this by leveraging the Python Standard Library module os. I leverage both the os.path. isdir and os.access methods associated with this module.

```
def ValidateDirectoryWritable(theDir):
```

I first check to see if in fact the directory string that the user provided exists. If the test fails then I raise an error within argparse and provide the message "Directory does not exist." This message would be provided to the user if the test fails.

```
# Validate the path is a directory
if not os.path.isdir(theDir):
raise argparse.ArgumentTypeError('Directory does not exist')
```

Next I validate that write privilege is authorized to the directory and once again if the test fails I raise an exception and provide a message.

```
# Validate the path is writable
if os.access(theDir, os.W_OK):
return theDir
else:
raise argparse.ArgumentTypeError('Directory is not writable')
```

Now that I have completed the implementation of the ParseCommandLine function, let us examine a few examples of how the function rejects improper command line arguments. In Figure 3.5, I created four improperly formed command lines:

(1) I mistyped the root directory as TEST_DIR instead of simply TESTDIR (2) I mistyped the –sha512 parameter as –sha521
(3) I specified two hash types –sha512 and –md5
(4) Finally, I did not specify any hash type

As you can see in each case, ParseCommandLine rejected the command.

In order to get the user back on track they simply have to utilize the –h or help

option as shown in Figure 3.6 to obtain the proper command line argument instructions.

WalkPath

Now let us walk through the WalkPath function that will traverse the directory structure, and for each file will call the HashFile function. I think you will be pleasantly surprised how simple this is.

```
C:\p-fish>python pfish.py --sha512  -d "c:\\p-fish\\TEST_DIR\\" -r "c:\\p-fish\\" -v
usage: Python file system hashing .. p-fish [-h] [-v]
                                            (--md5 | --sha256 | --sha512) -d
                                            ROOTPATH -r REPORTPATH
Python file system hashing .. p-fish: error: argument -d/--rootPath: Directory does not exist

C:\p-fish>
C:\p-fish>python pfish.py --sha521  -d "c:\\p-fish\\TESTDIR\\" -r "c:\\p-fish\\" -v
usage: Python file system hashing .. p-fish [-h] [-v]
                                            (--md5 | --sha256 | --sha512) -d
                                            ROOTPATH -r REPORTPATH
Python file system hashing .. p-fish: error: one of the arguments --md5 --sha256 --sha512 is required

C:\p-fish>
C:\p-fish>python pfish.py --sha512 --md5  -d "c:\\p-fish\\TESTDIR\\" -r "c:\\p-fish\\" -v
usage: Python file system hashing .. p-fish [-h] [-v]
                                            (--md5 | --sha256 | --sha512) -d
                                            ROOTPATH -r REPORTPATH
Python file system hashing .. p-fish: error: argument --md5: not allowed with argument --sha512

C:\p-fish>
C:\p-fish>python pfish.py -d "c:\\p-fish\\TESTDIR\\" -r "c:\\p-fish\\" -v
usage: Python file system hashing .. p-fish [-h] [-v]
                                            (--md5 | --sha256 | --sha512) -d
                                            ROOTPATH -r REPORTPATH
Python file system hashing .. p-fish: error: one of the arguments --md5 --sha256 --sha512 is required

C:\p-fish>_
```

FIGURE 3.5
Demonstration of ParseCommandLine.

```
C:\p-fish>python pfish.py -h
usage: Python file system hashing .. p-fish [-h] [-v]
                                            (--md5 | --sha256 | --sha512) -d
                                            ROOTPATH -r REPORTPATH

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         allows progress messages to be displayed
  --md5                 specifies MD5 algorithm
  --sha256              specifies SHA256 algorithm
  --sha512              specifies SHA512 algorithm
  -d ROOTPATH, --rootPath ROOTPATH
                        specify the root path for hashing
  -r REPORTPATH, --reportPath REPORTPATH
                        specify the path for reports and logs will be written

C:\p-fish>_
```

FIGURE 3.6
pfish –h command.

def WalkPath():

I first initialize the variable processCount in order to count the number of successfully processed files and I post a message to the log file to document the root path value.

processCount ¼ 0
errorCount ¼ 0
log.info('Root Path:'+ gl_args.rootPath)

Next I initialize the CSVWriter with the reportPath provided on the command line by the user. I also provide the hashType selected by the user so it can be included in the Header line of the CSV file. I will cover the CSVWriter class later in this chapter.

oCVS ¼ _CSVWriter(gl_args.reportPath+'fileSystemReport.csv', gl_hashType)

Create a loop that process all the files starting
at the rootPath, all sub-directories will also be
processed

Next I create a loop using the os.walk method and the rootpath specified by the user. This will create a list of file names that is processed in the next loop. This is done for each directory found within the path.

for root, dirs, files in os.walk(gl_args.rootPath):
# for each file obtain the filename and call the # HashFile Function

The next loop processes each file in the list of files and calls the function HashFile with the file name joined with the path, along with the simple file name for use by HashFile. The call also passes HashFile with access to the CVS writer so that the results of the hashing operations can be written to the CVS file.

for file in files:
fname¼ os.path.join(root, file)
result¼ HashFile(fname, file, oCVS)

if successful then increment ProcessCount
The process and error counts are incremented accordingly

```
if result is True:
processCount +¼ 1
# if not successful, the increment the ErrorCount
else:
errorCount +¼ 1
```

Once all the directories and files have been processed the CVSWriter is closed and the function returns to the main program with the number of successfully processed files.

```
oCVS.writerClose()
return(processCount)
```

HashFile

Below is the code for the HashFilefunction, it is clearly the longest for this program, but also quite simple and straightforward. Let us walk through the process.

```
def HashFile(theFile, simpleName, o_result):
```

For each file several items require validation before we attempt to hash the file.
(1) Does the path exist
(2) Is the path a link instead of an actual file
(3) Is the file real (making sure it is not orphaned)
For each of these tests there is a corresponding log error that is posted to the log file if failure occurs. If the file is bypassed the program will simply return to WalkFile and process the next file.

```
# Verify that the path is valid
if os.path.exists(theFile):
#Verify that the path is not a symbolic link
if not os.path.islink(theFile):
#Verify that the file is real
if os.path.isfile(theFile):
```

The next part is a little tricky. Even through our best efforts to determine the existence of the file, there may be cases where the file cannot be opened or read. This could be caused by permission issues, the file is locked or possibly corrupted. Therefore, I utilize the try methods while attempting to open and then read from the files. Note that I’m careful to open the file as read-only the “rb”

option. Once again if an error occurs a report is generated and logged and the program moves on to the next file.

```
try:
#Attempt to open the file
f ¼ open(theFile,'rb')

except IOError:
#if open fails report the error log.warning('Open Failed:'+ theFile) return

else:
try:
# Attempt to read the file
rd ¼ f.read()

except IOError:
# if read fails, then close the file and # report error
f.close()
log.warning('Read Failed:'+ theFile) return

else:
#success the file is open and we can #read from it
#lets query the file stats
```

Once the file has been successfully opened and verified that reading from the file is allowed, I extract the attributes associated with the file. These include owner, group, size, MAC times, and mode. I will include these in the record that is posted to the CSV file.

```
theFileStats ¼ os.stat(theFile)
(mode, ino, dev, nlink, uid, gid, size, atime, mtime, ctime) ¼ os.stat(theFile)

# Display progress to the user DisplayMessage("Processing File: " + theFile)

# convert the file size to a string fileSize ¼ str(size)
# convert the MAC Times to strings

modifiedTime ¼ time.ctime(mtime) accessTime ¼ time.ctime(atime) createdTime
¼ time.ctime(ctime)
```

convert the owner, group and file mode

ownerID ¼ str(uid) groupID ¼ str(gid) fileMode ¼ bin(mode)

Now that the file attributes have been collected the actual hashing of the file occurs. I need to hash the file as specified by the user (i.e., which one-way hashing algorithm should be utilized). I'm using the Python Standard Library module hashlib as we experimented with in Chapter 2.

#process the file hashes

if gl_args.md5:
#Calcuation the MD5
hash ¼ hashlib.md5()
hash.update(rd)
hexMD5 ¼ hash.hexdigest()
hashValue ¼ hexMD5.upper()

elif gl_args.sha256:
#Calculate the SHA256
hash¼hashlib.sha256()
hash.update(rd)
hexSHA256 ¼ hash.hexdigest() hashValue ¼ hexSHA256.upper()

elif gl_args.sha512:
#Calculate the SHA512
hash¼hashlib.sha512()
hash.update(rd)
hexSHA512 ¼ hash.hexdigest() hashValue ¼ hexSHA512.upper()

else:
log.error('Hash not Selected')
#File processing completed
#Close the Active File

Now that processing of the file is complete the file must be closed. Next I use the CSV class to write out the record to the report file and return successfully to the caller in this case WalkPath.

f.close()
# write one row to the output file

o_result.writeCSVRow(simpleName, theFile, fileSize, modifiedTime, accessTime, createdTime, hashValue, ownerID, groupID, mode)
return True

This section posts the warning messages to the log file relating to problems encountered processing the file.

else:
log.warning('['+ repr(simpleName) +', Skipped NOT a File'+']') return False

else:
log.warning('['+ repr(simpleName) +', Skipped Link NOT a File'+']')
return False

else:
log.warning('['+ repr(simpleName) +', Path does NOT exist'+']') return False

CSVWriter
The final code walk-through section I will cover in this chapter is the CSVWriter.AsI mentioned earlier, I created this code as a class instead of a function to make this more useful and to introduce you to the concept of classes in Python. The class only has three methods, the constructor or init, writeCSVRow, and writerClose. Let us examine each one.

class _CSVWriter:

The constructor or init method accomplishes three basic initializations: (1) Opens the output csvFile
(2) Initializes the csv.writer
(3) Writes the header row with the names of each column

If any failure occurs during the initialization an exception is thrown and a log entry is generated.

def __init__(self, fileName, hashType):
try:
# create a writer object and write the header row self.csvFile ¼

```
open(fileName,'wb')
self.writer ¼ csv.writer(self.csvFile,

delimiter ¼',', quoting¼csv.QUOTE_ALL) self.writer.writerow(
('File','Path','Size',
'Modified Time','Access Time','Created Time', hashType,'Owner','Group','Mode')
)
except:
log.error('CSV File Failure')
```

The second method writeCSVRow receives a record fromHashFileupon successful completion of each file hash. The method then uses the csv writer to actually place the record in the report file.

```
def writeCSVRow(self, fileName, filePath, fileSize, mTime, aTime, cTime,
hashVal, own, grp, mod):

self.writer.writerow( (fileName, filePath, fileSize, mTime, aTime, cTime,
hashVal, own, grp, mod))
```

Finally, the writeClose method, as you expect, simply closes the csvFile. def writerClose(self):
self.csvFile.close()
Full code listing pfish.py

```
#
# p-fish : Python File System Hash Program # Author: C. Hosmer # July 2013
# Version 1.0
#
import logging
import time
import sys
import _pfish
# Python Standard Library Logger
# Python Standard Library time functions # Python Library system specific
parameters # _pfish Support Function Module

if __name__¼¼'__main__':
PFISH_VERSION ¼'1.0'
```

```
# Turn on Logging
logging.basicConfig(filename¼'pFishLog.log',level¼logging.DEBUG,
format¼'%(asctime)s %(message)s')

# Process the Command Line Arguments _pfish.ParseCommandLine()

# Record the Starting Time startTime ¼ time.time()

# Record the Welcome Message
logging.info(')
logging.info('Welcome to p-fish version'+PFISH_VERSION +'...New Scan
Started')
logging.info(')
_pfish.DisplayMessage('Welcome to p-fish .. . version'+
PFISH_VERSION)

# Record some information regarding the system logging.info('System:'+
sys.platform)
logging.info('Version:'+ sys.version)

# Traverse the file system directories and hash the files filesProcessed ¼
_pfish.WalkPath()

# Record the end time and calculate the duration endTime ¼ time.time()
duration ¼ endTime startTime
logging.info('Files Processed:'+ str(filesProcessed)) logging.info('Elapsed
Time:'+ str(duration)+'seconds') logging.info(')
logging.info('Program Terminated Normally')
logging.info(')

_pfish.DisplayMessage("Program End")
```

Full code listing _pfish.py

```
#
# pfish support functions, where all the real work gets done #

# Display Message() # HashFile()
# ValidateDirectory() #

import os
```

```
import stat
import time
import hashlib
import argparse

import csv ParseCommandLine() WalkPath() class _CVSWriter
ValidateDirectoryWritable()

#Python Standard Library - Miscellaneous operating system interfaces
#Python Standard Library - functions for interpreting os results
#Python Standard Library - Time access and conversions functions
#Python Standard Library - Secure hashes and message digests
#Python Standard Library - Parser for commandline options, arguments
#Python Standard Library - reader and writer for csv files
import logging #Python Standard Library – logging facility

log ¼ logging.getLogger('main._pfish')

#
# Name: ParseCommand() Function
#
# Desc: Process and Validate the command line arguments # use Python
Standard Library module argparse #
# Input: none
#
# Actions:
#

#

# # Uses the standard library argparse to process the command line
establishes a global variable gl_args where any of the functions can
obtain argument information

def ParseCommandLine(): parser¼ argparse.ArgumentParser('Python file system
hashing .. p-fish')
parser.add_argument('- v','—verbose', help¼'allows progress messages to be
displayed', action¼'store_true')

# setup a group where the selection is mutually exclusive and required.
```

```
group¼ parser.add_mutually_exclusive_group(required¼True)
group.add_argument('--md5', help ¼'specifies MD5 algorithm',
action¼'store_true')
group.add_argument('--sha256', help ¼'specifies SHA256 algorithm',
action¼'store_true')
group.add_argument('--sha512', help ¼'specifies SHA512 algorithm',
action¼'store_true')

parser .add_argument('-d','--rootPath', type¼
ValidateDirectory, required¼True, help¼"specify the root path for hashing")
parser.add_argument('- r','--reportPath', type¼
ValidateDirectoryWritable, required¼True, help¼"specify the path for reports and
logs will be written")

# create a global object to hold the validated arguments, these will be available
then
# to all the Functions within the _pfish.py module

global gl_args
global gl_hashType
gl_args ¼ parser.parse_args()
if gl_args.md5:
gl_hashType ¼'MD5'

elif gl_args .sha256:
gl_hashType ¼'SHA256'
elif gl_args.sha512:
gl_hashType ¼'SHA512'
else:
gl_hashType ¼ "Unknown"
logging.error('Unknown Hash Type Specified')

DisplayMessage("Command line processed: Successfully")
return
# End
ParseCommandLine¼¼¼
```

Name: WalkPath() Function
#
Desc: Walk the path specified on the command line
use Python Standard Library module os and sys
Input: none, uses command line arguments
#
Actions:
Uses the standard library modules os and sys # to traverse the directory structure starting a root # path specified by the user. For each file discovered,

WalkPath
will call the Function HashFile() to perform the file
hashing
#

def WalkPath(): processCount ¼ 0 errorCount ¼ 0 oCVS ¼
_CSVWriter(gl_args.reportPath+'fileSystemReport.csv', gl_hashType)

Create a loop that process all the files starting
at the rootPath, all sub-directories will also be
processed

log.info('Root Path:'+ gl_args.rootPath)
for root, dirs, files in os.walk(gl_args.rootPath):
# for each file obtain the filename and call the HashFile Function for file in files:
fname¼ os.path.join(root, file)
result ¼ HashFile(fname, file, oCVS) # if hashing was successful then increment the ProcessCount if result is True:
processCount +¼ 1
if not successful, the increment the ErrorCount else:
ErrorCount +¼ 1
oCVS.writerClose()
return(processCount)
#End
WalkPath¼¼¼

Desc: Processes a single file which includes performing a hash of the

file
and the extraction of metadata regarding the file processed # use Python Standard Library modules hashlib, os, and sys
Input: theFile ¼ the full path of the file
simpleName ¼ just the filename itself
#
# Actions:
# Attempts to hash the file and extract metadata # Call GenerateReport for successful hashed files #
def HashFile(theFile, simpleName, o_result):

Verify that the path is valid if os.path.exists(theFile): #Verify that the path is not a symbolic link if not os.path.islink(theFile):
#Verify that the file is real if os.path.isfile(theFile):

try :
#Attempt to open the file
f ¼ open(theFile, 'r b')

except IOError :
#if open fails report the error log.warning('Open Failed:'+ theFile) return

else:
try:
# Attempt to read the file
rd ¼ f.read()
except IOError:

# if read fails, then close the file and report error
f.close()
log.warning('Read Failed:'+ theFile) return

else :
#success the file is open and we can read from it #lets query the file stats

theFileStats ¼ os.stat(theFile)
(mode, ino, dev, nlink, uid, gid, size, atime, mtime, ctime) ¼ os.stat(theFile)

```
#Print the simple file name
DisplayMessage("Processing File: " + theFile) # print the size of the file in Bytes
fileSize ¼ str(size)

#print MAC Times
modifiedTime ¼ time.ctime(mtime) accessTime ¼ time.ctime(atime)
createdTime ¼ time.ctime(ctime)

ownerID ¼ str(uid) groupID ¼ str(gid) fileMode ¼ bin(mode)

#process the file hashes

if gl_args .md5:
#Calcuation and Print the MD5 hash ¼ hashlib.md5()
hash.update(rd)
hexMD5¼ hash.hexdigest() hashValue ¼ hexMD5.upper()

elif gl_args .sha256:
hash¼hashlib.sha256()
hash.update(rd)
hexSHA256 ¼ hash.hexdigest() hashValue ¼ hexSHA256.upper()

elif gl_args .sha512:
#Calculate and Print the SHA512 hash¼hashlib.sha512()
hash.update(rd)
hexSHA512 ¼ hash.hexdigest() hashValue ¼ hexSHA512.upper()

else :
log.error('Hash not Selected') #File processing completed
#Close the Active File
print "¼¼¼
```

else :
log.warning('['+ repr(simpleName)+', Skipped NOT a File'+ ']')
return False

else :
log.warning('['+ repr(simpleName)+', Skipped Link NOT a File'+ ']')

return False
else:
log.warning('['+ repr(simpleName)+', Path does NOT exist'+ ']')

return False
End HashFile Function
¼¼¼

#
Name: ValidateDirectoryWritable Function
#
Desc: Function that will validate a directory path as # existing and writable. Used for argument validation only
Input: a directory path string
#
Actions:
if valid will return the Directory String
if invalid it will raise an ArgumentTypeError within

argparse
which will in turn be reported by argparse to the user

def ValidateDirectoryWritable(theDir):
# Validate the path is a directory
if not os.path.isdir(theDir):
raise argparse.ArgumentTypeError('Directory does not exist') # Validate the path is writable
if os.access(theDir, os.W_OK):
return theDir
else:
raise argparse.ArgumentTypeError('Directory is not writable')
#End ValidateDirectoryWritable ¼¼¼

if gl_args.verbose:
print(msg)
Results Presentation 83
return
#End
DisplayMessage¼¼½

#
Class: _CSVWriter
#
Desc: Handles all methods related to comma separated value operations
Methods constructor: Initializes the CSV File
writeCVSRow: # writerClose: Writes a single row to the csv file

Closes the CSV File
class _CSVWriter: def __init__(self, fileName, hashType):

try :
# create a writer object and then write the header row self.csvFile ¼
open(fileName, 'w b')
self.writer¼ csv.writer(self.csvFile, delimiter¼',', quoting¼csv.QUOTE_ALL)
self.writer.writerow(('F i l e', 'P a t h', 'S i z e','Modified Time', 'Access
Time','Created Time',hashType,'O w n e r','G r o u p','M o d e'))

except:
log.error('CSV File Failure')
def writeCSVRow(self, fileName, filePath, fileSize, mTime, aTime, cTime,
hashVal, own, grp, mod):
self.writer.writerow((fileName, filePath, fileSize, mTime, aTime, cTime,
hashVal, own, grp, mod))
writerClose(self):def
self.csvFile.close()
RESULTS PRESENTATION

Now that the walk-through has been completed and I have gone through a deep dive into the code, let us take a look at the results. In Figure 3.7, I executed the program with the following options:

C:\p-fish>Python pfish.py --md5 -d “c:\\p-fish\\TESTDIR\\” -r “c:\\p-fish\\” –v

The –v or verbose option was selected and the program displayed information regarding every file processed was selected as expected.
In Figure 3.8, I examine the c:\p-fish directory and discover that two files were created there, which are the two resulting files for the pfish.py.

fileSystemReport.csv1.
2. pFishLog.log

```
  -d ROOTPATH, --rootPath ROOTPATH
                        specify the root path for hashing
  -r REPORTPATH, --reportPath REPORTPATH
                        specify the path for reports and logs will be written

C:\p-fish>python pfish.py --md5 -d "c:\\p-fish\\TESTDIR\\" -r "c:\\p-fish\\" -v
Command line processed: Successfully
Wecome to p-fish ... version 1.0
Processing File: c:\\p-fish\\TESTDIR\hpwmdl21.dat
================================
Processing File: c:\\p-fish\\TESTDIR\hpwprl03.dat
================================
Processing File: c:\\p-fish\\TESTDIR\hpwprl04.dat
================================
Processing File: c:\\p-fish\\TESTDIR\hpwprl10.dat
================================
Processing File: c:\\p-fish\\TESTDIR\hpwprl11.dat
================================
Processing File: c:\\p-fish\\TESTDIR\Before and After\124.JPG
================================
Processing File: c:\\p-fish\\TESTDIR\Before and After\210.JPG
================================
Processing File: c:\\p-fish\\TESTDIR\Before and After\291.JPG
================================
Processing File: c:\\p-fish\\TESTDIR\Before and After\292.JPG
================================
Processing File: c:\\p-fish\\TESTDIR\Before and After\293.JPG
================================
```

FIGURE 3.7
Test run of pfish.py.

FIGURE 3.8
Result directory after pfish execution.

By choosing to leverage the Python csv module to create the report file Windows already recognizes it as a file that Microsoft Excel can view. Opening the file we see the results in Figure 3.9, a nicely formatted column report that can now manipulate with Excel (sort columns, search for specific values, arrange in date order, and examine each of the results). You notice that the hash value is in a column named MD5 that is labeled as such because I passed the appropriate heading value during the initialization of the csv.

FIGURE 3.9
Examining the Result File with Microsoft Excel.

```
2013-08-04 21:45:11,042
2013-08-04 21:45:11,042 Welcome to p-fish version 1.0 ... New Scan Started
2013-08-04 21:45:11,059
2013-08-04 21:45:11,059 System: win32
2013-08-04 21:45:11,059 Version: 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)]
2013-08-04 21:45:11,059 Root Path: c:\\p-fish\\TESTDIR\
2013-08-04 21:45:12,042 Files Processed: 82
2013-08-04 21:45:12,042 Elapsed Time: 0.999000072479 seconds
2013-08-04 21:45:12,042
2013-08-04 21:45:12,042 Program Terminated Normally
2013-08-04 21:45:12,042
```

FIGURE 3.10
Contents of the pFishLog file.

The generated pFishLog.log file results are depicted in Figure 3.10. As expected we find the welcome message, the details regarding the Windows environment, the root path that was specified by the user, the number of files processed, and the elapsed time of just under 1 s. In this example no errors were encountered and the program terminated normally.

Moving to a Linux platform for execution only requires us to copy two Python Files.
1. pfish.py
2. _pfish.py

Execution under Linux (Ubuntu version 12.04 LTS in this example) works without changing any Python code and produces the following results shown in Figures 3.11–3.13.

You notice that the pFishLog file under Linux has a number of warnings; this is due to lack of read access to many of the files within the /etc directory at the user privilege level I was running and due to some of the files being locked because they were in use.

CHAPTER REVIEW

In this chapter I created our first useable Python forensics application. The pfish.py program executes on both Windows and Linux platforms and through some ingenuity I only used Python Standard Library modules to accomplish this along with our own code. I also scratched the surface with argparse allowing us to not only parse the command line but also validate command line parameters before they were used by the application.

I also enabled the Python logger and reported events and errors to the logging system to provide a forensic record of our actions. I provided the user with the capability of selecting among the most popular one-way hashing algorithms and the program extracted key attributes of each file that was processed. I also leveraged the cvs module to create a nicely formatted output file that can be opened and processed by standard applications on both Windows and Linux systems. Finally, I implemented our first class in Python with many more to come.

FIGURE 3.11
Linux Command Line Execution.

FIGURE 3.12
Linux Execution Results pfish Result File.

FIGURE 3.13
Linux Execution Results pFishLog File.

SUMMARY QUESTIONS

1. If you wanted to add additional one-way hashing algorithms, which functions would you need to modify? Also, by using just the Python Standard Library what other one-way hashing algorithms are readily available.

2. If you wanted to eliminate the need of the two global variables how could you easily accomplish this by using classes? What function would you convert to a class and what methods would you need to create?

3. What other events or elements do you think should be logged? How would you go about doing that?
4. What additional columns would you like to see in the report and how would you obtain the additional information?
5. What additional information (such as Investigator name or case number) should be included in the log. How would you obtain that information?

LOOKING AHEAD
In Chapter 4, I will be continuing with the cookbook section by tackling searching and indexing of forensic data.

Additional Resources

Hosmer C. Using SmartCards and digital signatures to preserve electronic evidence. In: SPIE proceedings, vol. 3576. Forensic Technologies for Crime Scene and the Laboratory I. The paper was initially presented at the investigation and forensic science technologies symposium; 1998. Boston, MA, http://proceedings.spiedigitallibrary.org/proceeding.aspx? articleid¼974141 [01.11.1998].

Kim G. The design and implementation of tripwire: a file system integrity checker. Purdue ePubs computer science technical reports, 1993. http://docs.lib.purdue.edu/cstech/1084/. CHAPTER

Forensic Searching and Indexing Using Python 4

CHAPTER CONTENTS

Introduction 91
Keyword Context Search 93
How can This be Accomplished Easily in Python? 94
Fundamental Requirements 96
Design Considerations 96
Main Function 98
ParseCommandLine 98
SearchWords Function 98
PrintBuffer Functions 99
Logger 99
Writing the Code 99
Code Walk-Through 100
Examining Main—Code Walk-Through 100
Examining _p-Search Functions—Code Walk-Through 101
Examining ParseCommandLine 101
Examining ValidateFileRead(theFile) 103
Examining the SearchWords Function 103
Examining the PrintBuffer Function 106
Results Presentation .....

107
Indexing 109
Coding isWordProbable 112
p-Search Complete Code Listings 114
p-Search.py 114
_p-Search.py 116
Chapter Review 122
Summary Questions 123
Additional Resources 123

INTRODUCTION

Searching is certainly one of the pillars of forensic investigation and today a search is only as good as the investigator running it. Knowing what to search for, where to search for it (in a file, in deleted files, slack space, e-mail files, a database, or application data), and then interpreting the search results requires both experience and knowledge of criminal behavior. Over the past several years we have seen the emergence of indexing of evidence and we have also seen frustration with the performance of indexing methods. The Python programming language has several built-in language mechanisms along with Standard Library modules that will assist in both searching and indexing.

© 2014 Elsevier Inc. All rights reserved.

There are many reasons for searching and indexing, and I want to make sure we examine more than just simple keywords, names, hashes, etc. So what is the underlying reason or rationale why we search? Referencing fictional works or characters is always a risk when writing a book like this, but in this case I believe it works. Most of us have read some if not all of the works of Sir Arthur Conan Doyle's Sherlock Holmes novels and short stories. In these fictional accounts, Holmes uses both deductive and inductive reasoning to create theories about the crime and profiles of the suspects, victims, and even bystanders.

Deductive reasoning takes no risk, in other words if you have verifiable facts along with a valid argument the conclusion must be true. For example:

1. All men are mortal
2. Socrates was a man
Therefore, Socrates was mortal.
Inductive reasoning on the other hand takes risks and uses probabilities. For example:

1. John always leaves for the office at 6:00 AM
2. He punches in between 7:30 and 7:45 AM each day
3. John left home today promptly at 6:00 AM verified by a surveillance camera and punched in at work at 7:42 AM.
Therefore, John could not have committed a murder at 7:05 AM that occurred over 3 miles from the office.

The basis of these hypotheses whether based on deductive or inductive reasoning is facts. Therefore, I believe the reason to search and index is to discover facts that we then use to create theories and hypothesis. I know this may seem brutally obvious, but then again we sometimes narrow in on a theory before we have any trustworthy facts. This is especially true in the digital world where we can be quick to judge as the saying goes “because I saw it on the Internet” when in fact the Internet or digital data have been manipulated to skew the facts.

Seasoned investigators understand this and are typically very thorough in their investigation of digital crime scenes. I have had the privilege of working with many such investigators and they are tenacious in validation of digital evidence. I cannot tell you how many times I have been asked “are you sure about that?” or “can you prove that” or even “how else could this have happened?” Finally, we need to be conscious in our searching to discover both inculpatory and exculpatory evidence and make sure both types of facts are recorded before we create theories or hypothesis.

Inculpatory evidence supports a defined hypothesis
Exculpatory evidence is contradictory to a defined hypothesis

The question then becomes what facts do we wish to discover? Fundamentally, the answer is who, what, where, when, and how or to be slightly more specific, here are small set of examples.

You might be wondering about the “Why” of the search. We tend not to think about why as in many cases this is generated by the investigator, through experience, intuition, and deep understanding regarding the circumstances that surround the case.

• What documents exist and what is their content or relevance? When were they created, modified, last accessed, when and how many times were they printed? Were the documents stored or written to a flash device or stored in the cloud? Where did they originate?
• What multimedia files exist and where did they originate? For example, were they downloaded from the Internet or recorded or snapped by the suspect or victim. If so, what camera or recording device was used?
• Who is the suspect(s) we are investigating and what do we know about them? Do we have a photograph, phone numbers, e-mail addresses, home address, where do they work, when and where have they traveled, do they own a vehicle(s), if so what make, model, year, and plate number? Do they have an alias (physical world or cyber world)?
• Who are their known associates, what do we know about them?

I think you get the idea, the questions that we might ask during a search are vast with some being very specific and others more general. In this chapter, we certainly cannot solve all of these issues, but we can scratch the surface and create Python programs that will assist us now and be able to be evolved in the future.

KEYWORD CONTEXT SEARCH

Keyword searching with context is the ability to perform a search of a data object like a disk image or a memory snapshot, and discover keywords or phrases that relate to a specific context or category. For example, I might want to perform a search of a disk image for each occurrence of the following list of street names for the drug cocaine:

blow, C, candy, coke, do a line, freeze, girl, happy dust, mama coca, mojo, monster, nose, pimp, shot, smoking gun, snow, sugar, sweet stuff, white powder, base, beat, blast, casper, chalk, devil drug, gravel, hardball, hell, kryptonite, love, moonrocks, rock, scrabble, stones, and tornado.

How can this be accomplished easily in Python?

In order to tackle keyword searching with Python we must first address several simple issues. First, how should we store the search words or phrases? Python has several built-in data types that are ideally suited to handle such lists each with their own unique capabilities and rules. The basic types are sets, lists, and dictionaries.My preference would be to create a set to hold the search words and phrases. Sets in

Python provide a nice way of dealing with this type of data mainly due to the fact that they automatically eliminate duplicates, they are trivial to implement, and they can easily be tested for members.

The code below demonstrates the basics. I start by initializing an empty set named searchWords. Next, the code opens and then reads the file narc.txt that contains narcotic-related words and phrases. I of course include exception handling for the file operations to ensure we successfully opened and then can read each line from the file. Next, I print out the list of search words and phrases that were loaded line by line (notice one word or phrase per line) and then search the list for the word “kryptonite” and either print “Found word” if the word exists in the search words list or “not found” if the word does not exist. One fine point, I specify line.strip() when adding the words to the set searchWords. This strips the newline characters that exist at the end of each line.

```
import sys
searchWords ¼ set()
try:

fileWords ¼ open('narc.txt')
for line in fileWords:
searchWords.add(line.strip())

except:
print("file handling error)"
sys.exit()

print(searchWords)
if ('kryptonite'in searchWords):
print("Found word)"
else:
print("not found)"
```

Executing the code produces the following output:
Starting with the contents of the set:

{ 'shot','devil drug"do a line"scrabble"casper"hell"kryptonite','mojo"blow' 'stones tornado' 'white powder' 'smoking gun' 'happy dust' 'gravel' 'hardball' 'moonrocks"monster"beat"snow sugar"coke"rock"base"blast"pimp"sweet stuff"candy"chalk"nose"mama coca"freeze girl'}

Then printing out the results of the search for kryptonite produces Found word
This simple code then solves two fundamental challenges:

(1) Reading a list into a set from a file
(2) Searching a set to determine if an entry exits in a set. Next, we have to solve the problem of extracting words and phrases from the target of a search. Almost always this requires that a binary file or stream be parsed. These binary data could be a memory snapshot, a disk image or a snapshot of network data. The key is the data would be either unformatted or formatted based on one of hundreds of standards, some open and some proprietary with text and binary data intertwined. Therefore, if we would like to develop a simple search tool that could extract and compare text strings without regard to the format or content, how might we accomplish this?

The first challenge would be to define what Python data object we would use to hold these binary data. For this I have chosen the Python object bytearray as this fits our criteria exactly. As the name implies bytearrays are simply a series of bytes (a byte being and unsigned 8-bit value) starting at the zero array element and extending to the end of the array. I am going to take advantage of a couple of Python specialties, for example, loading a file into a bytearray, the core code to do so is only two lines.

The first line opens a file for reading in binary mode. Whereas the second line reads the contents of the file into a bytearray named baTarget. You notice that I prefix the object Target with “ba” this simply makes the code more readable and anyone will immediately recognize that the object is of type bytearray. Also, note this code is just an illustration. In the real program, we have to check the return values from the function calls and make sure there are no errors, I will illustrate that technique when we get to the actual program.

targetFile ¼ open('file.bin','rb')

baTarget ¼ bytearray(targetFile.read())

One of the immediate questions you might have is, “how do I know how many bytes were loaded into the bytearray?” This is handled just like any other Python object by using the len function.

sizeOfTarget¼len(baTarget)
In Figure 4.1, we can see the object baTarget that was created and also the sizeOfTarget that was determined of baTarget using the len function.

Variable	Value
locals	<dict 0x202e930; len=7>
baTarget	<huge bytearray 0x1f255f0>
baTargetCopy	<huge bytearray 0x1f255f0>
fileWords	<closed file 'c:\\pytest\\narc.txt', mode 'r' at 0x01FAED30>
line	'tornado\n'
searchWords	<set 0x1fc6120; len=27>
sizeOfTarget	77650
targetFile	<closed file 'c:\\pytest\\capture.raw', mode 'rb' at 0x01FAED88>

FIGURE 4.1
Snapshot of stackdata displaying the baTarget object and the size in bytes of the baTarget bytearray.

Fundamental requirements
Now that I have defined the new language and data elements we intend to use for our simple text search, I want to define the basic requirements of the program as shown in Table 4.1.

Design considerations
Now that I have defined the basic requirements for the application, I need to factor in the design considerations. For the search program, I will be using Python Standard Library modules and the specialized code we developed for parsing the target file. Table 4.2 covers the basic mapping of requirements to modules and functions that will be utilized.

Next, I want to define the overall design of the search capability. As with the p-fish program we developed in Chapter 3, I will be using the command line and command line arguments to specify the inputs to the program. For the search results, I will be directing the output to standard output using some specialized methods developed to render the data for easy interpretation. Finally, I will be

using the built-in logging functions to complete the forensic nature of the search (Figure 4.2).

Turning to the internal structure, I have broken the program down into four main components. The Main program, ParseCommandLine function, the actual SearchWords function, PrintBuffer functions for outputting the results, and logger (note logger is actually the Python logger module), that is utilized by the major functions of p-search. I briefly describe the operation of each below and during the code

Table 4.1 Basic Requirements Requirement Requirement number name
SRCH-001 Command Line Arguments

SRCH-002 Log
SRCH-003 Performance SRCH-004 Output

SRCH-005 Error Handling Short description

Allow the user to specify a file containing keywords (ASCII in this example)
Allow the user to specify a file containing a binary file to search
Allow the user to specify verbose output regarding the progress of search
The program must produce a forensic audit log The program should perform the search efficiently The program should identify any of the keywords found in the binary file and provide a Hex/ASCII printout, the offset where the keyword was found along with the surrounding values in order to provide context
The application must support error handling and logging of all operations performed. This will include a textual description and a timestamp

Table 4.2 Standard Library Mapping
Requirement Design considerations

User Input (001) User input for keyword and target file

Performance (003) Selecting the right language and library objects to process the data is important when designing a search method

Output (004) The search results must be provided in a meaningful manner. In other words, mapping the identified keywords to the offset in the file along with pre- and postdata that surround the identified keywords

Logging and error Errors may occur during the handling (002 and processing of the keyword 005) and target file. Therefore,

handling these potential errors must be rigorous Library selection

I will be using argparse from the Standard Library module to obtain input from the user
I will be using Python sets and bytearrays to handle the data related to searching which will increase performance and make the search process extensible I will be using the Standard Library print function to accomplish this, but instead of writing the result to a file directly, I am going to write the data to standard out, the user can then pipe the output if they wish to a file or into other programs or functions
The Python Standard Library includes aloggingfacility which I can leverage to report any events or errors that occur during processing

User

Standard out

Program Arguments
p-search

Event
and
error log

p-search context diagram FIGURE 4.2
p-search context diagram.

p-search internal structure Standard out
ParseCommandLine

Program Arguments Keywords
target file PrintBuffer HEX ASCII printout

Main ASCII
heading

SearchWords foundWords

Offset buffer

Events
logPath logger LogRecords events
FIGURE 4.3
p-search internal structure.
PrintHeading

Event
and
error log

walk-through section a more detailed line by line explanation of how each function operates is provided (Figure 4.3).

Main function
The purpose of the Main function is to control the overall flow of this program. For example, within Main, I setup the Python logger, I display startup and completion messages, and keep track of the time. In addition, Main invokes the command line parser and then launches the SearchWords function. Once SearchWords completes, Main will log the completion and display termination messages to the user and the log.

ParseCommandLine
In order to provide smooth operation of p-search, I leverage the ParseCommandLine to not only parse but also validate the user input. Once completed, information that is needed by the core search method will be available via the parser.

SearchWords function
SearchWords is the core of the p-search program. The objective is to make this function as quick and accurate as possible. In this example, we are only searching for ASCII text strings or words. This provides a huge advantage and we can set the search up in a way that is quick and effective. The algorithm performs the search in two passes over the baTarget bytearray. The first pass converts any byte that is not an alpha character to a zero. The second pass collects sequences of alpha characters. If the consecutive alpha character sequence is greater than or equal to MIN_WORD and less than or equal to MAX_WORD before encountering a zero, the characters are collected and then compared to the set of keywords. Note MIN_WORD and MAX_WORD will be defined as constants. In the future, they could be command line arguments to the search.

PrintBuffer functions
The PrintBuffer function is called by SearchWords whenever a keyword match is discovered. Then the offset within the file and Hex/ASCII display are sent to standard out.

logger

The built-in Standard Library logger provides us with the ability to write messages to a log file associated with p-search. The program can write information messages, warning messages, and error messages. Since this is intended to be a forensic application, logging operations of the program are vital. You can expand the program to log additional events in the code; they can be added to any of the _p-searchfunctions.

Writing the code

Once again, I broke the code into source files p-search.py and _p-search.py, where _p-search.py provides the support and heavy lifting functions of p-search. In Figure 4.4 you can see the WingIDE environment along with the p-search program running. The upper left corner displays the local and global variables. In the upper

FIGURE 4.4
WingIDE p-search execution.

center panel you get a glimpse of the program output showing the match of the word “tornado” along with the detailed Hex/ASCII representation of the output. On the far right, the files associated with the program are listed. Finally, in the bottom portion of the screen you can see the source code for the program. In the

following section, we will walk-through the important code elements to disclose the detailed method and approach I have taken.

CODE WALK-THROUGH

I will be inserting dialog as I discuss each code section. The code walk-through will give you an in-depth look at all the codes associated with the program. I will be first walking through each of the key functions and then will provide you with a complete listing of both p-search.py and _p-search.py.

Examining Main—code walk-through

```
import logging
import time
import _psearch
```

For the main program we need to import logging to record forensic events and time to calculate the duration of program execution both from the Python Standard Library. We also import our _psearch module that contains the supporting and core functions of p-search

```
if __name__ ¼¼'__main__':
P-SEARCH_VERSION ¼'1.0'
# Turn on Logging
logging.basicConfig(filename¼'pSearchLog.log',level¼logging.DEBUG,
format¼'%(asctime)s %(message)s')
```

Next I invoke command line parser to obtain the program arguments passed in by the user.

```
# Process the Command Line Arguments
_psearch.ParseCommandLine()
```

I then setup logging to occur and send the startup events to the logger. log ¼ logging.getLogger('main._psearch')
log.info("p-search started)"

I record the start time of execution in order calculate duration when the search is complete.

```
# Record the Starting Time
startTime ¼ time.time()
```

Next, I invoke the SearchWords function. Note that this functions is contained in the _psearch module thus I need to prefix the call with the module name _psearch.

```
# Perform Keyword Search
_psearch.SearchWords()
```

I record the end time and calculate the duration and make the final entries in the log.

```
# Record the Ending Time
endTime ¼ time.time()
duration¼ endTime - startTime

logging.info('Elapsed Time:'+ str(duration) +'seconds') logging.info('')
logging.info('Program Terminated Normally')
```

Examining _p-search functions—code walk-through

The remaining functions are contained in the _p-search.py file. The file starts by importing the needed modules. These include argparse for processing command line arguments. The os module is for handling file I/O operations. Finally, logging is used for forensic logging capabilities.

```
import argparse
import os
import logging
```

Examining ParseCommandLine

For the p-search program, we only require two parameters from the user—the full path name of the file containing the keywords to search along with the full path name of the file we wish to search. The verbose option is optional and when it is not provided the program messages will be suppressed (Table 4.3).

As you can see the ParseCommandLine prepares for three arguments verbose, keywords and srchTarget. Keywords and search target are required parameters and they are both validated by the function ValidateFileName().

```
def ParseCommandLine():
parser¼ argparse.ArgumentParser('Python Search')
parser.add_argument('-v','--verbose', help¼"enables printing of program
messages", action¼'store_true')
```

parser.add_argument('-k','--keyWords', type¼ ValidateFileRead, required¼True, help¼"specify the file containing search words)" parser.add_argument('-t','--srchTarget', type¼ ValidateFileRead, required¼True, help¼"specify the target file to search)”

Table 4.3 p-search command line argument definition Option Description Notes

-v Verbose, if this option is specified then any calls to the DisplayMessage() function will be displayed to the standard output device, otherwise the program will run silently

-k Keyword, this allows the user to specify the path of the file containing the keywords

-t Target, this allows the user to specify the path of the file to be searched The file must exist and must be readable or else the program will abort

The file must exist and must be readable or else the program will abort

Next we establish a global variable to hold results of parse_args operations. Any invalid entries will automatically abort processing.

```
global gl_args
gl_args ¼ parser.parse_args()
DisplayMessage("Command line processed: Successfully)"
return
```

One of the nice features of argparse is the ability to include a help message with each entry. Then whenever the user specifies -h on the command line argparse automatically assembles the appropriate response. This allows the developer to provide as much information as possible to assist the user in providing the correct response, so be verbose. Figure 4.5 depicts the operation of the -h option for p-search.

FIGURE 4.5
p-search execution using only the -h or help option.

Examining ValidateFileRead(theFile)

This function supports the ParseCommandLineand parser.parse_args()by validating any filenames provided by the user. The function first ensures that the file exists and then verifies that the file is readable. Failing either of these checks will result in the program aborting and will raise the appropriate exception.

```
def ValidateFileRead(theFile):
# Validate the path is a valid
if not os.path.exists(theFile):
raise argparse.ArgumentTypeError('File does not exist') # Validate the path is readable

if os.access(theFile, os.R_OK):
return theFile
else:
raise argparse.ArgumentTypeError('File is not readable')
```

Examining the SearchWords function

The core of p-search is the SearchWords function which compares the keywords provided by the user with the contents of the target file specified. We will take a deep dive line by line of SearchWords.

```
def SearchWords():
# Create an empty set of search words
```

As I mentioned in the design considerations, I decided to use a Python set object to hold the keywords to search. To do this I simply assign searchWords as a set object.

searchWords ¼ set()

Next, I need to load the keywords from the file specified by the user in the command line arguments. I do this using the ‘try, except, finally’ method to ensure we trap or capture any failures.

# Attempt to open and read search words
try:
fileWords ¼ open(gl_args.keyWords)
Once we successfully open the keywords file I process each line striping the words from the line and adding it to the searchWords set.
for line in fileWords: searchWords.add(line.strip())
except:
Any exceptions encountered are logged and the program is aborted
log.error('Keyword File Failure:'+ gl_args.keyWords) sys.exit()
finally:
Once all the lines have been processed successfully, the file is closed.
fileWords.close()
# Create Log Entry Words to Search For Next, I write entries into the forensic log to record the words that are to be included in the search.

log.info('Search Words')
log.info('Input File:'+gl_args.keyWords)
log.info(searchWords)

Attempt to open and read the target file # and directly load the file into a bytearray Now that I have the keywords to search for, I will read the target file provided and store the data into a Python byte array object.

try:
targetFile ¼ open(gl_args.srchTarget,'rb')
baTarget ¼ bytearray(targetFile.read())

except:
Any exceptions will be caught, logged and the program will exit.

log.error('Target File Failure:'+ gl_args.srchTarget) sys.exit()
finally:
targetFile.close()
I record the size of the target file on success.
sizeOfTarget¼ len(baTarget)
# Post to log I also post this information to the forensic log file.
log.info('Target of Search:'+ gl_args.srchTarget) log.info('File Size:'+str(sizeOfTarget))

I'm going to modify baTarget by substituting zero for all not alpha characters. In order to ensure that I can display the original contents of the target file, I make a copy.

baTargetCopy ¼ baTarget
Search Loop
step one, replace all non characters with zero's

The first search step is to traverse the target and replace all non alpha characters with zero to make the search of words much easier and faster. The loop below starts at the beginning of the file and performs the replacement.

for i in range(0, sizeOfTarget):
character ¼ chr(baTarget[i])
if not character.isalpha():

baTarget[i] ¼ 0
step # 2 extract possible words from the bytearray # and then inspect the search word list

Now that the bytearray baTarget only contains valid alpha characters and zeros I can count any consecutive alpha character sequences and if the sequence meets the size characteristics defined I can check the collected sequences of characters against the smaller keyword list

create an empty list of not found items

To be thorough, I'm creating a list of notFoundcharacter sequences that meet our criteria of sequences of characters, but do not match the keywords specified by the user.

notFound¼ []

Now I begin the actual search. I simply check each byte and if it qualifies as a character I increase the count by one. If I encounter a zero, I stop the count and see if I have encountered enough consecutive characters to qualify as a possible word. If I don't then I set the count back to zero and continue the search. If on the other hand the count meets the criteria for the minimum and maximum number of characters I have a possible word.

cnt ¼ 0
for i in range(0, sizeOfTarget):
character ¼ chr(baTarget[i])
if character.isalpha():
cnt +¼ 1
else:
if (cnt >¼ MIN_WORD and cnt <¼ MAX_WORD):
newWord ¼ “”

If I have a sequence that meets the criteria I need to collect the characters that I have now skipped. One of the tricks is to not bother collecting or building possible strings until you meet the criteria. Once the criteria is met I backtrack and pull the consecutive letters together and store them in the variable newWord. I use the newWord variable to search the set of keywords.

for z in range(i-cnt, i):
newWord ¼ newWord þ chr(baTarget[z])

To search the set of keywords stored in the set seachWords, I simply write the one line test. This is the power of properly choosing the correct object type for storing the keywords.

if (newWord in searchWords):

If I get a hit in the searchWord list, I call the PrintBuffer function with the details of the hit which includes the word I found, the offset in the buffer, the saved unmodified buffer, where to start printing the result and where to stop.

PrintBuffer(newWord, i-cnt, baTargetCopy, iPREDECESSOR_ SIZE, WINDOW_SIZE)

print
else:
notFound.append(newWord)
cnt ¼ 0
else:
cnt ¼ 0

After I have processed the complete bytearray, I also print out the notFound list I created, this will give the investigator additional possible words to consider that were not in the keyword list.

PrintNotFound(notFound) # End of SearchWords Function

Examining the PrintBuffer function
Most of us that investigate cybercrime for a living use Hex editors or viewers quite often. The PrintBuffer function is a very simple Hex and ASCII viewer written in Python. It is simple to ensure that it will run on almost any platform (Windows, Linux, Mac, and even mobile devices). The trick is to not get fancy, but instead just provide the facts. Let us take a look how this can be accomplished with just a few lines of Python code.

def PrintBuffer(word, directOffset, buff, offset, hexSize):

The inputs to the function are as follows:
word: The alpha string that was identified
directOffset: The offset in the target file to the beginning of the alpha string buff: The actual buffer
offset: The offset where the Hex/ ASCII Window should start hexSize: What is the size in bytes that should be displayed
I start by printing the basic facts, the word and offset where the string was found.

print "Found: "+ word + " At Address: ", print "%08x " % (directOffset)
Next, I print the heading which prints the following heading. Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0 F ASCII PrintHeading() Next, I just need to print the Hex and ASCII values found in the buffer starting that the specified offset. The outer loop continues in 16 value chunks since each line

will hold the next 16 values. Since we are printing this out in Hex this makes the most sense.

```
for i in range(offset, offset+hexSize, 16):
for j in range(0,17):
```

The inside loop then needs to print each of the next 16 bytes in hex. I start by printing the offset value in hex leaving the appropriate amount of same. Note also, that I use the syntax %08x which allows for an 8 digit Hex address zero filled. This should be plenty for this demonstration application.

```
if (j ¼¼ 0):
print "%08x " % i,
else:
byteValue ¼ buff[i+j]
print "%02x " % byteValue,
print " ",
```

Next we repeat this loop but this time we print the ASCII character representation instead of the Hex values.

```
for j in range (0,16):
byteValue ¼ buff[i+j]
```

To avoid any special characters we only print the ASCII printable set and then replace the other values with a period.

```
if (byteValue >¼ 0x20 and byteValue<¼ 0x7f):
print "%c" % byteValue,
else:
print'.',
print
# End Print Buffer
```

RESULTS PRESENTATION

Now that the walk-through has been completed and I have gone through a deep dive into the code, let us take a look at the results. Figure 4.6 is a snapshot of the directory that contains all the requisite files. This includes the two Python program files p-search.py and _p-search.py. It includes the narc.txt file that contains the search words. Capture.raw is a file containing binary data that we wish to search. pSearchLog.log is the forensic log file output from p-search and finally results.txt is a file containing the standard output results that were piped to a file for easy viewing.

Figure 4.7 lists the file containing the drug-related keywords that we intend to search for.
Figure 4.8 is the direct output of the program starting with the command line entry which specified that the results be piped to the file results.txt. Also included is the results.txt file. You can see that we identified three hits in the target file, they include sugar, tornado, and moonrocks. You also see the Hex/ASCII printout generated by the program along with the unidentified words at the bottom.

FIGURE 4.6
Execution test directory for p-search.

FIGURE 4.7
Keyword file dump.

Finally in Figure 4.9, you see the contents of the log file generated by p-search that includes the start message, the search word file, and contents. Also included is the name and size of the target file capture.raw. Finally, you see the elapsed time for execution in seconds and the program termination message.

To round out this example and demonstrate the portability, I have included two additional screenshots running the unmodified p-search program on both a Linux machine (Figure 4.10) and iMac (Figure 4.11).

```
C:\Windows\system32\cmd.exe

C:\PYTEST>python p-search.py -k narc.txt -t capture.raw > results.txt

C:\PYTEST>more results.txt
Found: sugar At Address:  00000406
Offset        00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F        ASCII
-------------------------------------------------------------------------------------------
000003eb      00  77  48  00  6a  75  6c  69  6f  00  79  61  68  6f  6f  00        . . w H . j u l i o . y a h o o .
000003fb      63  6f  6d  00  00  7a  00  6f  66  00  73  75  67  61  72  00        . c o m . . z . o f . s u g a r .
0000040b      00  00  00  00  00  00  00  00  00  66  69  00  00  00  00  00        . . . . . . . . . . f i . . . . .
0000041b      00  00  00  00  00  00  00  00  00  6d  66  00  00  00  00  00        . . . . . . . . . . m f . . . . .
0000042b      00  00  00  00  00  00  00  6b  00  51  00  00  00  00  00  00        . . . . . . . . k . Q . . . . . .
0000043b      00  00  00  00  00  00  00  00  00  00  00  00  55  55  4e  00        . . . . . . . . . . . . U U N .
0000044b      00  00  00  00  42  00  00  00  00  00  00  00  00  00  00  00        . . . . . B . . . . . . . . . . .
0000045b      00  00  00  00  00  00  00  00  00  00  68  73  68  6d  00  6e        . . . . . . . . . . . h s h m . n

Found: tornado At Address:  000007ad
Offset        00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F        ASCII
-------------------------------------------------------------------------------------------
00000794      79  00  61  6f  6c  00  63  6f  6d  00  6e  65  65  64  00  74        n y . a o l . c o m . n e e d . t
000007a4      6f  00  73  65  65  00  61  00  74  6f  72  6e  61  64  6f  00        t o . s e e . a . t o r n a d o .
000007b4      00  00  00  00  00  00  00  00  00  00  00  52  00  00  68  68        . . . . . . . . . . . R . . h h
000007c4      00  4e  00  51  4b  44  00  00  00  00  00  00  00  00  00  00        h . N . Q K D . . . . . . . . . .
000007d4      00  00  66  00  00  00  00  00  00  00  73  6b  75  00  00  48        . . . f . . . . . . . s k u . . H
000007e4      66  00  6f  00  76  00  00  00  00  64  59  6b  00  00  00  00        H f . o . v . . . . d Y k . . . .
000007f4      00  65  00  00  00  74  6c  00  00  00  00  00  58  6a  74  00        . . e . . . t l . . . . . X j t .
00000804      00  00  63  00  00  00  00  00  56  00  00  00  00  5a  00  00        . . . c . . . . . V . . . . Z . .

Found: moonrocks At Address:  00000e10
Offset        00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F        ASCII
-------------------------------------------------------------------------------------------
00000df9      63  6f  6d  00  68  61  76  65  00  79  6f  75  00  73  65  65        . c o m . h a v e . y o u . s e e
00000e09      6e  00  61  6e  79  00  6d  6f  6f  6e  72  6f  63  6b  73  00        e n . a n y . m o o n r o c k s .
00000e19      00  00  00  00  00  00  00  00  00  00  4f  00  00  00  4f  00        . . . . . . . . . . O . . . O .
00000e29      00  00  63  00  00  00  00  00  5a  00  00  00  57  00  00  00        . . . c . . . . . Z . . . W . . .
00000e39      00  00  00  00  00  00  00  00  00  00  59  00  00  00  00  00        . . . . . . . . . . Y . . . . .
00000e49      00  00  00  00  00  00  58  00  00  00  00  00  00  00  53  6d        . . . . . . X . . . . . . S m .
00000e59      00  00  4e  00  00  00  00  00  00  00  00  6e  00  00  00  00        . . . N . . . . . . . n . . . .
00000e69      00  72  5a  00  00  00  00  45  00  00  00  00  00  00  00  00        . . r Z . . . . E . . . . . . . .

------- Possible Words Identified, but not found -------
JFIF,  Photoshop,  JFIF,  File,  written,  Adobe,  Photoshop,  Adobe,  GWgw,  AQaq,  GWgw,  julio,  yahoo,  ugar
,  hshm,  johnny,  need,  ornado
,  QJdF,  gmail,  have,  seen,  oonrocks
,  vINC,  lonV,  kmmn,  XICC,  PROFILE,  HLino,  mntrRGB,  acspMSFT,  sRGB,  cprt,  desc,  lwtpt,  bkpt,  rXYZ,  gXYZ,
  rTRC,  gTRC,  bTRC,  text,  Copyright,  Hewlett,  Packard,  Company,  desc,  sRGB,  sRGB,  desc,  http,  http,  desc,
pace,  sRGB,  desc,  Reference,  Viewing,  Condition,  Reference,  Viewing,  Condition,  view,  meas,  curv,  File,  wri
  qYZOD,  CIYX,  ouno,  anDq,  uvxXCW,  ROhX,  FpId,  wavt,  zGdP,  KMEDH,  MGIB,  CuqN,  MpVP,  yWLj,  yhJR,  LJUQf,  h
  dhmHg,  with,  GLnG,  UJmQO,  imZd,  tchS,  tidz,  wIKca,  lzTe,  qFWx,  KhRP,  REAZ,  sEXx,  iRWg,  pGYk,  fwqz,  AJS
  qRMP,  YDxf,  qyij,  ixmCm,  FhmIR,  zwNFX,  CZyq,  lyCk,  aeVG,  IsrP,  iEjPI,  wTkt,  JIRm,  cgshH,  NqWD,  qJmO,  O
 CMfl,  aoGiJ,  OzKqPd,  KfKla,  lRyNq,  CduX,  jtvC,  UVRz,  ARzT,  iSEm,  BRHW,  RORENi,  uiEtQ,  rYIq,  rVPo,  nViNQ,
a,  ojCj,  RpfD,  mkpXO,  eUJG,  yTiV,  PZtp,  ziYso,  ipTLid,  Suxo,  FvaD,  KgVT,  kuBS,  UPhzkm,  HMAKF,  IJJA,  aikz
ty,  USytZ,  hLeV,  ZYtk,  zhHJ,  OgCm,  xvFq,  nKny,  OdPfg,  UgHSf,  AQIR,  iiPA,  OuLZV,  YSuR,  xFVc,  hRdd,  mjiN,

C:\PYTEST>
```

FIGURE 4.8
p-search sample execution.

```
C:\Windows\system32\cmd.exe

C:\PYTEST>more pSearchLog.log
2013-08-27 21:16:30,755 p-search started
2013-08-27 21:16:30,756 Search Words
2013-08-27 21:16:30,756 Input File: narc.txt
2013-08-27 21:16:30,756 set(['stones', 'shot', 'scrabble', 'powder', 'casper', 'hell', 'sugar', 'mojo', 'blow', 'devil',
ardball', 'moonrocks', 'monster', 'beat', 'coke', 'base', 'blast', 'pimp', 'candy', 'chalk', 'nose', 'rock'])
2013-08-27 21:16:30,756 Target of Search: capture.raw
2013-08-27 21:16:30,757 File Size: 77650
2013-08-27 21:16:30,973 Elapsed Time: 0.218999862671 seconds
2013-08-27 21:16:30,974
2013-08-27 21:16:30,974 Program Terminated Normally

C:\PYTEST>_
```

FIGURE 4.9
Log file contents post execution.

INDEXING

Indexing provides the investigator with a different look at the data and potential

evidence contained within a file, disk image, memory snapshot, or network trace. Instead of determining what keywords to look for, we want p-search to provide us

```
chet@PythonForensics: ~/Desktop/Python Samples/p-search
chet@PythonForensics:~/Desktop/Python Samples/p-search$ python p-search.py -k narc.txt -t capture.raw
Found: sugar At Address:  00000406
Offset        00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F        ASCII
-----------------------------------------------------------------------------------------------
000003eb      00  77  48  00  6a  75  6c  69  6f  00  79  61  68  6f  6f  00        . . w H . j u l i o . y a h o o .
000003fb      63  6f  6d  00  00  7a  00  6f  66  00  73  75  67  61  72  00        . c o m . . z . o f . s u g a r .
0000040b      00  00  00  00  00  00  00  00  00  66  69  00  00  00  00  00        . . . . . f i . . . . .
0000041b      00  00  00  00  00  00  00  00  00  6d  66  00  00  00  00  00        . . . . . m f . . . . .
0000042b      00  00  00  00  00  00  00  6b  00  51  00  00  00  00  00  00        . . . . . k . Q . . . . .
0000043b      00  00  00  00  00  00  00  00  00  00  00  00  55  55  4e  00        . . . . . U U N .
0000044b      00  00  00  00  42  00  00  00  00  00  00  00  00  00  00  00        . . . . . B . . . . . .
0000045b      00  00  00  00  00  00  00  00  00  00  68  73  68  6d  00  6e        . . . . . h s h m . n

Found: tornado At Address:  000007ad
Offset        00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F        ASCII
-----------------------------------------------------------------------------------------------
00000794      79  00  61  6f  6c  00  63  6f  6d  00  6e  65  65  64  00  74        n y . a o l . c o m . n e e d . t
000007a4      6f  00  73  65  65  00  61  00  74  6f  72  6e  61  64  6f  00        t o . s e e . a . t o r n a d o .
000007b4      00  00  00  00  00  00  00  00  00  00  00  52  00  00  68  68        . . . . . R . . h h
000007c4      00  4e  00  51  4b  44  00  00  00  00  00  00  00  00  00  00        h . N . Q K D . . . . . .
000007d4      00  00  66  00  00  00  00  00  00  00  73  6b  75  00  00  48        . . . f . . . . . s k u . . H
000007e4      66  00  6f  00  76  00  00  00  00  64  59  6b  00  00  00  00        H f . o . v . . . . d Y k . . . .
000007f4      00  65  00  00  00  74  6c  00  00  00  00  00  58  6a  74  00        . . e . . . t l . . . . . X j t .
00000804      00  00  63  00  00  00  00  00  56  00  00  00  00  5a  00  00        . . . c . . . . . V . . . . Z . .

Found: moonrocks At Address:  00000e10
Offset        00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F        ASCII
-----------------------------------------------------------------------------------------------
00000df9      63  6f  6d  00  68  61  76  65  00  79  6f  75  00  73  65  65        . c o m . h a v e . y o u . s e e
00000e09      6e  00  61  6e  79  00  6d  6f  6f  6e  72  6f  63  6b  73  00        e n . a n y . m o o n r o c k s .
00000e19      00  00  00  00  00  00  00  00  00  00  4f  00  00  00  4f  00        . . . . . O . . . O .
00000e29      00  00  63  00  00  00  00  00  5a  00  00  00  57  00  00  00        . . . c . . . . . Z . . . W . . .
00000e39      00  00  00  00  00  00  00  00  00  00  59  00  00  00  00  00        . . . . . Y . . . . .
00000e49      00  00  00  00  00  00  58  00  00  00  00  00  00  53  6d  00        . . . . . X . . . . . S m .
00000e59      00  00  4e  00  00  00  00  00  00  00  00  6e  00  00  00  00        . . . N . . . . . n . . . .
00000e69      00  72  5a  00  00  00  00  45  00  00  00  00  00  00  00  00        . . r Z . . . . E . . . . . .

------- Possible Words Identified, but not found -------
JFIF, Photoshop, JFIF, File, written, Adobe, Photoshop, Adobe, GWgw, AQaq, GWgw, julio, yahoo, ugar, hshm, john
ny, need, ornado, QJdF, gmail, have, seen, oonrocks, vlMC, lonV, kmmn, XICC, PROFILE, HLino, mntrRGB, acspMSFT,
  sRGB, cprt, desc, lwtpt, bkpt, rXYZ, gXYZ, bXYZ, dmnd, pdmdd, vued, view, lumi, meas, tech, rTRC, gTRC, bTR
C, text, Copyright, Hewlett, Packard, Company, desc, sRGB, sRGB, desc, http, http, desc, Default, colour, space
, sRGB, Default, colour, space, sRGB, desc, Reference, Viewing, Condition, Reference, Viewing, Condition, view,
meas, curv, File, written, Adobe, Photoshop, Adobe, EeIj, hMYp, vacpj, qYZOD, CIYX, ouno, anDq, uvxXCW, RQhX,
FpId, wavt, zGdP, KMEDH, MGIB, CuqN, MpVP, yWLj, yhJR, LJUQf, hOBX, PRTI, McfS, sSpaHE, vYZo, oRoI, XShx, qSN
I, dhnHg, with, GLnG, UJmQO, imZd, tchS, tidz, wIKca, lzTe, qFWx, KhRP, REAZ, sEXx, iRWg, pGYk, fwqz, AJSI,
FQRZq, xUEw, yFcqCw, UNwk, TdTd, lRMOB, vLyn, qRMP, YDxf, qyij, ixmCm, FhmIR, zwNFX, CZyq, lyCk, aeVG, IsrP,
iEjPI, wTkt, JIRm, cgshW, NqWD, qJmO, OOZk, rChS, sKqk, qJAJ, Ambj, zcIyn, BcKu, Wmku, CMfl, aoGiJ, OzKqPd, K
fKla, lRyNq, CduX, jtvC, UVRz, ARzT, iSEm, BRHW, ROREN, uiEtQ, rYIq, rVPo, nViNQ, ptQrm, TfEs, Fnaq, Cifs, tp
aq, OlkW, zrCW, ZxwCa, ojCj, RpfD, mkpXO, eUJG, yTiV, PZtp, ziYso, ipTLid, Suxo, FvaD, KqVT, kuBS, UPhzkm, HM
AKF, IJJA, aikz, xWNt, zIWP, ilwd, gjUB, nKGm, hAIq, LbJR, XFwty, USytZ, hLeV, ZYtk, zhHJ, OgCm, xvFq, nKny,
 OdPfg, UgHSf, AQIR, iiPA, OuLZV, YSuR, xFVc, hRdd, mjiN, PCbjU, eMvRs, hSHP, llYq,
```

FIGURE 4.10
Execution of p-search running on Ubuntu Linux 12.04 LTS.

with a list of all the words that meet the minimum definition of a string, but do not match any of the defined key words. However, the current approach is lacking two key pieces of data. The first is simple, I need to include the offset of where each string was found in the target file. The second problem is a bigger challenge, that being to only print words that are likely to be legitimate words. Finally, it would be nice if the words were provided in a sorted list.

To handle the more difficult problem of identifying strings that have a high

probability of being words, I have devised an approach that works exceptionally well and is fast. I depict the basic concept in Figure 4.12. I first process known words with an algorithm that produces a numeric weight based on the internal characteristics of each word.

Once a numeric weight is calculated for the word, the weights are sorted and any duplicates are removed. The key here is to process a large list of words. For this example, I used a dictionary containing over 600,000 words. This produces a matrix

```
janet-hosmers-imac:p-search janet$ python p-search.py -k narc.txt -t capture.raw -m matrix.txt
Found: sugar At Address:  00000406
Offset        00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F        ASCII
-----------------------------------------------------------------------------------------------
000003eb      00  77  48  00  6a  75  6c  69  6f  00  79  61  68  6f  6f  00        . . w H . j u l i o . y a h o o .
000003fb      63  6f  6d  00  00  7a  00  6f  66  00  73  75  67  61  72  00        . c o m . . z . o f . s u g a r .
0000040b      00  00  00  00  00  00  00  00  00  66  69  00  00  00  00  00        . . . . . . . . . . f i . . . . .
0000041b      00  00  00  00  00  00  00  00  00  6d  66  00  00  00  00  00        . . . . . . . . . . m f . . . . .
0000042b      00  00  00  00  00  00  00  6b  00  51  00  00  00  00  00  00        . . . . . . . . k . Q . . . . . .
0000043b      00  00  00  00  00  00  00  00  00  00  00  00  55  55  4e  00        . . . . . . . . . . . . U U N .
0000044b      00  00  00  00  42  00  00  00  00  00  00  00  00  00  00  00        . . . . . B . . . . . . . . . . .
0000045b      00  00  00  00  00  00  00  00  00  00  68  73  68  6d  00  6e        . . . . . . . . . . . h s h m . n

Found: tornado At Address:  000007ad
Offset        00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F        ASCII
-----------------------------------------------------------------------------------------------
0000079t      79  00  61  6f  6c  00  63  6f  6d  00  6e  65  65  64  00  74        n y . a o l . c o m . n e e d . t
000007at      6f  00  73  65  00  61  00  74  6f  72  6e  61  64  6f  00  00        t o . s e e . a . t o r n a d o .
000007bt      00  00  00  00  00  00  00  00  00  00  00  52  00  00  68  68        . . . . . . . . . . . R . . h h
000007ct      00  4e  00  51  4b  44  00  00  00  00  00  00  00  00  00  00        h . N . Q K D . . . . . . . . . .
000007dt      00  00  66  00  00  00  00  00  00  00  73  6b  75  00  00  48        . . . f . . . . . . . s k u . . H
000007et      66  00  6f  00  76  00  00  00  00  64  59  6b  00  00  00  00        H f . o . v . . . . d Y k . . . .
000007ft      00  65  00  00  00  74  6c  00  00  00  00  00  58  6a  74  00        . . e . . . t l . . . . . X j t .
00000804      00  00  63  00  00  00  00  00  56  00  00  00  00  5a  00  00        . . . c . . . . . V . . . . Z . .

Found: moonrocks At Address:  00000e10
Offset        00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F        ASCII
-----------------------------------------------------------------------------------------------
00000df9      63  6f  6d  00  68  61  76  65  00  79  6f  75  00  73  65  65        . c o m . h a v e . y o u . s e e
00000e09      6e  00  61  6e  79  00  6d  6f  6f  6e  72  6f  63  6b  73  00        e n . a n y . m o o n r o c k s .
00000e19      00  00  00  00  00  00  00  00  00  00  4f  00  00  00  4f  00        . . . . . . . . . . . O . . . O .
00000e29      00  00  63  00  00  00  00  00  5a  00  00  00  57  00  00  00        . . . c . . . . . Z . . . W . . .
00000e39      00  00  00  00  00  00  00  00  00  00  59  00  00  00  00  00        . . . . . . . . . . Y . . . . .
00000e49      00  00  00  00  00  00  58  00  00  00  00  00  00  53  6d  00        . . . . . . . X . . . . . . S m .
00000e59      00  00  4e  00  00  00  00  00  00  00  00  6e  00  00  00  00        . . . N . . . . . . . . n . . . .
00000e69      00  72  5a  00  00  00  00  45  00  00  00  00  00  00  00  00        . . r Z . . . . E . . . . . . . .

Index of All Words
---------------------
['adobe', 470]
['adobe', 494]
['adobe', 10042]
['adobe', 10066]
['aogij', 46360]
['colour', 7624]
['colour', 7681]
['company', 7257]
['condition', 7752]
['condition', 7807]
['copyright', 7222]
['default', 7612]
['default', 7669]
['gmail', 3572]
['hewlett', 7241]
```

FIGURE 4.11
Execution of p-search running on iMac.

Unique sorted weights

Words
Word Word weights$_{\text{weighng}}$
algorithm

FIGURE 4.12
Diagram of word weighting approach.
r u n n i n g
c u n n i n g
g u n n i n g
d u n n i n g

Used in weighng calculaon FIGURE 4.13
Weighted characteristics illustration.

of weights that can be used to compare strings found in raw data (using the same weighting algorithm used to create the original matrix) to the matrix. If the calculated weight matches one found in the matrix, I consider this a possible word, if not the string is discarded. Figure 4.13 provides an illustration of how this works. Each of the core letters contained in the word (ignoring the leading character) are used in the weighted calculation.

Since many words, at least in the English language, are derived from other words, this eliminates the need for having an all-encompassing dictionary of every possible word. In addition, if words are misspelled or newly created or derived, it is likely the weighting system vs. a direct match will provide a more liberal interpretation of a word being probable.

The importance of approximate searches and indexes can be vital. In the Casey Anthony case, for example, investigators missed probative Internet search history data because they looked for various search terms which *they* spelled correctly. She, however, had misspelled several search terms; while Google corrected the spelling during the search, it did not fix the misspelling in the Internet search history.

CODING isWordProbable
In order to create a method isWordProbable(word), I created a class named Matrix; let us take our typical deep dive into this new class.
class class_Matrix:
I start out by declaring a new set object named the weighted matrix.
weightedMatrix ¼ set()
Coding isWordProbable 113

When the class is instantiated, the object will load the values contained in the matrix into the weighted matrix set. To handle this I simply added a new command line argument -m or --theMatrix to allow the user to specify a matrix file. I take the same precautions as you have seen before to trap any errors that might occur during the file handling operations.

```
def __init__(self):
try:
fileTheMatrix ¼ open(gl_args.theMatrix,'rb') for line in fileTheMatrix:

value¼ line.strip()
self.weightedMatrix.add(int(value,16)) except:
log.error('Matrix File Error:'+ gl_args.theMatrix) sys.exit()
finally:
fileTheMatrix.close()
return
```

Next I define the method isWordProbable which will calculate the weight of the

passed string and perform a lookup in the weightedMatrix to determine if we have a match. Note I first check to see if theWord passed is of minimum size.

```
def isWordProbable(self, theWord):

if (len(theWord) < MIN_WORD):
return False
else:
BASE ¼ 96
wordWeight ¼ 0
```

This is the core weight calculation and produces an unsigned long integer value name wordWeight.

```
for i in range(4,0,-1):
charValue¼ (ord(theWord[i]) - BASE) shiftValue ¼ (i-1)*8
charWeight ¼ charValue << shiftValue wordWeight ¼ (wordWeight |
charWeight)
```

Once I have calculated the word weight I just check to see if the weight exists in the weighted matrix.

```
if ( wordWeight in self.weightedMatrix):
return True
else:
return False
```

To properly incorporate this into the program the following code is all that is needed.

The initialization is simple, I create an object named wordCheck (which executes the initialization code of the Matrix to load in the weighed values. I also, create a list named indexOfWords where I will store the probable words that we find.

```
wordCheck¼ class_Matrix() indexOfWords ¼ []
...
...
```

Next, I embed the following code within the existing search loop to evaluate each string as a probable word. If wordCheck returns True, then I add the word to the indexOfWords list. Note that this list has two elements per entry: 1) the string newWord and 2) the offset from the beginning of the target object where it was found.

```
if wordCheck.isWordProbable(newWord):
indexOfWords.append([newWord, i-cnt])
```

Finally, I added a method to print the indexOfWords list at the end of the search that includes both the word and offset. I also sorted the list prior to print in order to make it alphabetical.

```
def PrintAllWordsFound(wordList):
print "Index of All Words"
print "————————————————————————————————————"
wordList.sort()
for entry in nfList:
print entry
print "————————————————————————————————————"
print
return
```

From an execution perspective you can see the results of the combined search and index program in Figure 4.14.

P-SEARCH COMPLETE CODE LISTINGS p-search.py

```
#
# p-search : Python Word Search
# Author: C. Hosmer
# August 2013
# Version 1.0
#
# Simple p-search Python program
#
```

```
Found: moonrocks At Address:  00000e10
Offset        00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F        ASCII
-----------------------------------------------------------------------------------------------
00000df9      63  6f  6d  00  68  61  76  65  00  79  6f  75  00  73  65  65        . c o m . h a v e . y o u . s e e
00000e09      6e  00  61  6e  79  00  6d  6f  6f  6e  72  6f  63  6b  73  00        e n . a n y . m o o n r o c k s .
00000e19      00  00  00  00  00  00  00  00  00  00  4f  00  00  00  4f  00        . . . . . O . . . O .
00000e29      00  00  63  00  00  00  00  00  5a  00  00  00  57  00  00  00        . . . c . . . . . Z . . . W . . .
00000e39      00  00  00  00  00  00  00  00  00  00  59  00  00  00  00  00        . . . . . Y . . . . .
00000e49      00  00  00  00  00  00  58  00  00  00  00  00  00  53  6d  00        . . . . . X . . . . . S m .
00000e59      00  00  4e  00  00  00  00  00  00  00  00  6e  00  00  00  00        . . . N . . . . . n . . . .
00000e69      00  72  5a  00  00  00  00  45  00  00  00  00  00  00  00  00        . . r Z . . . . E . . . . . .

Index of All Words
---------------------
['adobe', 470]
['adobe', 494]
['adobe', 10042]
['adobe', 10066]
['aogij', 46360]
['colour', 7624]
['colour', 7681]
['company', 7257]
['condition', 7752]
['condition', 7807]
['copyright', 7222]
['default', 7612]
['default', 7669]
['gmail', 3572]
['hewlett', 7241]
['hlino', 6881]
['johnny', 1936]
['julio', 1008]
['moonrocks', 3600]
['packard', 7249]
['photoshop', 24]
['photoshop', 476]
['photoshop', 10048]
['profile', 6868]
['qyzod', 11694]
['reference', 7734]
['reference', 7789]
['roren', 51038]
['space', 7631]
['space', 7688]
['sspahe', 22967]
['sugar', 1030]
['tornado', 1965]
['viewing', 7744]
['viewing', 7799]
['written', 459]
['written', 10031]
['yahoo', 1014]
---------------------
```

FIGURE 4.14
p-search execution with indexing capability.

Read in a list of search words
Read a binary file into a bytearray
Search the bytearray for occurrences of any specified search words # Print a HEX/ ASCII display localizing the matching words # Print out a list of possible words identified that didn't match
Definition of a word. a word for this example is an uninterrupted

sequence of
4 to 12 alpha characters
#

import logging
import time
import _psearch

if __name__ ¼¼'__main__':
PSEARCH_VERSION ¼'1.0'
Turn on Logging

logging.basicConfig(filename¼'pSearchLog.log',level¼logging.DEBUG, format¼'%(asctime)s %(message)s')
# Process the Command Line Arguments _psearch.ParseCommandLine()
log ¼ logging.getLogger('main._psearch') log.info("p-search started")
# Record the Starting Time startTime ¼ time.time() # Perform Keyword Search
_psearch.SearchWords()

Record the Ending Time
endTime ¼ time.time()
duration ¼ endTime startTime

logging.info('Elapsed Time:'+ str(duration)+'seconds') logging.info(')
logging.info('Program Terminated Normally')
_p-search.py

#
psearch support functions, where all the real work gets done
Display Message() ParseCommandLine() # ValidateFileRead() # Matrix (class)
#
import argparse

import os
import logging ValidateFileWrite()

Python Standard Library Parser for command-line options, arguments

Standard Library OS functions
Standard Library Logging
functions

log ¼ logging.getLogger('main._psearch')

#Constants
MIN_WORD ¼ 5
MAX_WORD ¼ 15

PREDECESSOR_SIZE ¼ 32 WINDOW_SIZE¼ 128
Minimum word size in bytes

Maximum word size in bytes # Values to print before match found # Total values to dump when match found

Name: ParseCommand() Function
Desc: Process and Validate the command line arguments # use Python Standard Library module argparse
Input: none
#
Actions:
Uses the standard library argparse to process the command

line
#
def ParseCommandLine():

parser¼ argparse.ArgumentParser('Python Search')

parser .add_argument('- v','--verbose', help¼"enables printing of additional program messages", action¼'store_true')
parser.add_argument('- k','--keyWords', type¼ ValidateFileRead, required¼True, help¼"specify the file containing search words") parser.add_argument('- t','--srchTarget', type¼ ValidateFileRead, required¼True, help¼"specify the target file to search") parser.add_argument('- m','--theMatrix', type¼ ValidateFileRead, required¼True, help¼"specify the weighted matrix file")

global gl_args gl_args ¼ parser.parse_args() DisplayMessage("Command line processed: Successfully") return

End Parse Command Line

#
Name: ValidateFileRead Function
#
Desc: Function that will validate that a file exists and is readable
Input: A file name with full path
#

Actions:
if valid will return path
#
if invalid it will raise an ArgumentTypeError within

argparse
# which will inturn be reported by argparse to the user #
def ValidateFileRead(theFile):

# Validate the path is a valid
if not os.path.exists(theFile):
raise argparse.ArgumentTypeError('File does not exist') # Validate the path is readable
if os.access(theFile, os.R_OK):
return theFile
else:
raise argparse.ArgumentTypeError('File is not readable')
#End ValidateFileRead
¼¼¼

```
# Name SearchWords()
#
# Uses command line arguments
#
# Searches the target file for keywords #

def SearchWords():
# Create an empty set of search words searchWords ¼ set()

# Attempt to open and read search words try:
fileWords ¼ open(gl_args.keyWords)
for line in fileWords:
searchWords.add(line.strip())
except:
log.error('Keyword File Failure:'+ gl_args.keyWords)
sys.exit()
finally:
fileWords.close()
# Create Log Entry Words to Search For
log.info('Search Words')
log.info('Input File:'+gl_args.keyWords) log.info(searchWords)

# Attempt to open and read the target file # and directly load into a bytearray

try : targetFile ¼ open(gl_args.srchTarget, 'r b') baTarget ¼
bytearray(targetFile.read())

except :
log.error('Target File Failure:'+ gl_args.srchTarget) sys.exit()

finally:
targetFile.close()

sizeOfTarget¼ len(baTarget) # Post to log

log.info('Target of Search:'+ gl_args.srchTarget) log.info('File
Size:'+str(sizeOfTarget))

baTargetCopy ¼ baTarget wordCheck ¼ class_Matrix()
```

Search Loop
step one, replace all non characters with zero's

for i in range (0, sizeOfTarget): character ¼ chr(baTarget[i]) if not character.isalpha():

baTarget[i] ¼ 0

step # 2 extract possible words from the bytearray # and then inspect the search word list
create an empty list of probable not found items

indexOfWords ¼ []

cnt ¼ 0
for i in range(0, sizeOfTarget):
character ¼ chr(baTarget[i])
if character.isalpha():
cnt +¼ 1
else:
if (cnt >¼ MIN_WORD and cnt <¼ MAX_WORD): newWord ¼ ""
for z in range(i-cnt, i):
newWord ¼ newWord + chr(baTarget[z]) newWord ¼ newWord.lower()
if (newWord in searchWords):
PrintBuffer(newWord, i-cnt, baTargetCopy,
i-PREDECESSOR_SIZE, WINDOW_SIZE)
indexOfWords.append([newWord, i-cnt])
cnt ¼ 0
print
else:
if wordCheck.isWordProbable(newWord): indexOfWords.append([newWord, i-cnt])
cnt ¼ 0
else:
cnt ¼ 0

PrintAllWordsFound(indexOfWords) return
End of SearchWords Function

```
#
# Print Hexidecimal / ASCII Page Heading
#
def PrintHeading():

print ("Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ASCII")
print("————————————————————————————————————————————————————————————————————————————————————————")

return
# End PrintHeading

#
# Print Buffer
#
# Prints Buffer contents for words that are discovered
# parameters
# 1) Word found
# 2) Direct Offset to beginning of the word
# 3 buff The bytearray holding the target
# 4) offset starting position in the buffer for printing
# 5) hexSize, size of hex display windows to print
#
def PrintBuffer(word, directOffset, buff, offset, hexSize):

print "Found: "+ word + " At Address: ",
print "%08x "%(directOffset)
PrintHeading()
for i in range(offset, offset+hexSize, 16):
for j in range(0,16):
if (j ¼¼ 0):

print "%08x " % i,
else:
byteValue ¼ buff[i+j]
print "%02x " % byteValue,
print " ",
for j in range (0,16):
byteValue ¼ buff[i+j]
```

```
if (byteValue >¼ 0x20 and byteValue <¼ 0x7f):
print "%c" % byteValue,
else:
print'.',
print
return

# End Print Buffer

#
# PrintAllWordsFound #

def PrintAllWordsFound(wordList):
print "Index of All Words" print
"————————————————————————————————————" wordList.sort() for entry in
wordList: print entry
print "————————————————————————————————————" print
return
#End PrintAllWordsFound

#
# Class Matrix
#
# init method, loads the matrix into the set # weightedMatrix
#
# isWordProbable method
# 1) Calculates the weight of the provided word # 2) Verifies the minimum
length
# 3) Calculates the weight for the word # 4) Tests the word for existence in the
matrix # 5) Returns true or false
#
class class_Matrix:

weightedMatrix ¼ set()

def __init__(self): try:
fileTheMatrix ¼ open(gl_args.theMatrix, 'r b') for line in fileTheMatrix:

value¼ line.strip()
```

```
self.weightedMatrix.add(int(value,16)) except:
log.error('Matrix File Error:'+ gl_args.theMatrix) sys.exit()
finally:
fileTheMatrix.close()
return def isWordProbable(self, theWord):

if (len(theWord) < MIN_WORD): return False
else:
BASE ¼ 96
wordWeight ¼ 0

for i in range (4,0,1):
charValue¼ (ord(theWord[i]) BASE) shiftValue ¼ (i-1)*8
charWeight ¼ charValue << shiftValue wordWeight ¼ (wordWeight |
charWeight)

if ( wordWeight in self.weightedMatrix): return True
else:
return False

## End Class Matrix
```

CHAPTER REVIEW

In this chapter, I put to use new Python language elements including sets and bytearrays and I expanded the use of lists and classes. I also provided examples of how to open and read files containing a diverse set of values such as keywords, raw binary, and hexadecimal values and work with those directly within Python bytearrays, lists, and sets. I demonstrated how leveraging the right language elements can lead to simple, readable code that accomplishes our objectives. As a result, p-search is a useable application that provides investigators with the capability to search for keywords and

generate an index of words found in documents, disk images, memory snapshots, and even network traces.

In addition, I once again demonstrated the interoperability of Python code by running the sample unaltered code on Windows, Linux, and Mac producing the same results.

SUMMARY QUESTIONS

1. What are the advantages and disadvantages of the built-in language types (lists, sets, and dictionary).
2. Explain the advantages of using a set to hold a list of keywords.
3. How does Python's ability to deal with a broad set of data (binary, text, and integers) differ from other languages?
4. Many modern files, disk images, memory snapshots, and network traces are going to contain Unicode data not only simple ASCII. Modify p-search to provide the ability to search and index data that contains a mixture of simple ASCII and Unicode.
5. Expand the matrix to include a broader set of weighted values that include foreign language, proper names, places, and vocabulary from other domains such as medical terminology.

Additional Resources

The Complete Sherlock Holmes: All 4 Novels and 56 Short Stories. Deluxe edition. Bantam Classics; October 1, 1986.
Python Programming Language—Official Website, Python.org. http://www.python.org.
The Python Standard Library. http://docs.python.org/2/library/.

CHAPTER

Forensic Evidence Extraction (JPEG and TIFF) 5

CHAPTER CONTENTS

Introduction 125
The Python Image Library 126
Before Diving Straight In 128
PIL Test-Before Code 131
Determining the Available EXIF TAGS 131
Determining the Available EXIF GPSTAGS 133
p-ImageEvidenceExtractor Fundamental Requirements 137
Design Considerations 137
Code Walk-Through .....

137
Main Program 141
Class Logging 141
cvsHandler 141
Command Line Parser 141
EXIF and GPS Handler 141
Examining the Code 141
Main Program 141
EXIF and GPS Processing 144
Logging Class 148
Command Line Parser 149
Comma Separated Value (CSV) Writer Class 150
Full Code Listings 151
Program Execution 158
Chapter Review .....
159
Summary Questions .....
159
Additional Resources .....
163

INTRODUCTION

Simple and complex digital data structures offer significant opportunities to collect and extract valuable evidence. Files have simple metadata that define modified, access, and created times along with file ownership, file size, and other attributes that label the file as read-only, system, or archive. This information is easy to retrieve and examine with common desktop tools. Memory snapshots contain both structured and unstructured data that can be carved and reconstructed to reveal running processes and threads, recent user activity, network data, and even user typed passwords and cryptographic keying material.

© 2014 Elsevier Inc. All rights reserved.

Other complex files structures, such as the JPEG files that include EXIF data (The Exchangeable Image File Format), contain a plethora of potentially valuable information about the image itself. This can include the camera type used to take the photograph, the time and date the photo was taken, and literally

hundreds of individual data elements associated with the photograph. With the advent of Facebook, Instagram, Twitter, Flickr, and other social media services, the sharing of digital photographs has been weaved into the fabric of our culture. Today, over a billion people carry smart phones and tablets that have built-in cameras capable of taking high-quality photographs and movies that can be shared instantly across the globe using the aforementioned social network applications, text messaging, or good old-fashioned e-mail.

In addition, everyone from the Weather Channel to law enforcement investigators use digital photography to provide insight and understanding of physical events. The Federal Communications Commission (FCC) has adopted new rules that include mandatory improvements to the accuracy of the location information transmitted when cell phone-based 911 calls are made. Specifically, E911 rules require wireless service providers to deliver the latitude and longitude of the caller within 50-300 m [FCC]. These requirements have driven smart phone manufactures to include Global Positioning System (GPS) chips in their handsets along with high-quality cameras that can automatically embed location and time data directly into digital photographs. Consequently these innovations have delivered potentially valuable evidence including “what: the content of the photo,” “when: the photo was snapped,” and “where: the photo was taken.”

Based on this, I wanted to delve deeper into the extraction of evidence from digital photographs, specifically those photos taken with smart phones that include EXIF data. I will focus most of my attention on the EXIF data associated with JPEG and TIFF images, mainly because the preponderance of photos generated today from smart phones support EXIF embedding.

Many of you may have already examined the contents of digital photographs and the EXIF information using Hex editors and other primitive tools, and as a result require an upgraded prescription from the optometrist afterword. To help us in peeling back the onion-like layers of digital photographs, I will be adding a new library that will do most of the heavy lifting for us—namely thePython Image Library(PIL). The PIL provides a great deal of capabilities to not only extract information from images but also provides capabilities to manipulate them. I am going to focus only on the data extraction of EXIF data using the PIL, but once you learn how to properly use the library you can use the PIL for other image processing activities [Python Image Library].

The Python Image Library

The first step in using the PIL is to install the library. If you are running Python on Windows platform, you are in luck as a simple installer is available as shown in Figure 5.1.

FIGURE 5.1
Downloading Python Image Library for Windows.

FIGURE 5.2
Windows installation Wizard for the Python Image Library.

Once you have successfully downloaded the proper version for your Python installation (I am using PIL version 1.1.7 for Python 2.7), you simply execute the downloaded file and accept the defaults provided by the Wizard as shown in Figure 5.2. (This is a great example of why I choose to utilize Python 2.x for the book, as you can see the PIL library has not yet been converted to work with Python 3.x. Once this is done, the programs in this chapter can be easily updated to work with Python 3.x.)

If you are working with Linux or Mac, the steps for library installation are readily available on the Web. Here is one excellent example for Ubuntu 12.04 LTS: https:// gist.github.com/dwayne/3353083 (Figure 5.3).

Before diving straight in
Before I dive right into the PIL, I need to introduce the built-in Python Dictionary type. Up to this point we have used the lists and sets structures within Python in several examples. In some cases, structures with a bit more flexibility

and capability are required. More specifically, since the PIL uses dictionaries, we need to better understand their operation.

As you can see in Code Script 5.1, each entry in the dictionary has two parts, first the key which is followed by the value. For example, the key “fast” has an associated value of “Porsche.”

installing-pil.md Markdown

Steps to install PIL

```
$ sudo apt-get install python-imaging
$ sudo apt-get install libjpeg-dev libfreetype6 libfreetype6-dev zlib1g-dev

$ sudo ln -s /usr/lib/`uname -i`-linux-gnu/libfreetype.so /usr/lib/
$ sudo ln -s /usr/lib/`uname -i`-linux-gnu/libjpeg.so /usr/lib/
$ sudo ln -s /usr/lib/`uname -i`-linux-gnu/libz.so /usr/lib/

$ pip install PIL
```

References:

- http://askubuntu.com/questions/156484/how-do-i-install-python-imaging-library-pil
- http://www.sandersnewmedia.com/why/2012/04/16/installing-pil-virtualenv-ubuntu-1204-precise-pangolin/

FIGURE 5.3
Installing Python Image Library on Ubuntu 12.04 LTS.

CODE SCRIPT 5.1 CREATING A DICTIONARY

```
>>> dCars ¼ {"fast": "Porsche", "clean": "Prius", "expensive": "Rolls"}
>>> print dCars
{'expensive':'Rolls','clean':'Prius','fast':'Porsche'}
```

You might have noticed that when I printed the contents of the dictionary, the order has changed. This is due to the fact that the key is converted to a hash value when stored in order to both save space and to improve performance when searching or indexing the dictionary. If you want the dictionary printed in a specific order you can use the following code, which extracts the dictionary items and then sorts the items by the key.

CODE SCRIPT 5.2 PRINTING A SORTED DICTIONARY

```
>>> dCarsItems ¼ dCars.items()
```

```
>>> dCarsItems.sort()
>>> print dCarsItems
[('clean','Prius'), ('expensive','Rolls'), ('fast','Porsche')]
```

What if the dictionary is large and you wish to find all the keys? Or once you find the keys you wish to extract the value of a specific key?
CODE SCRIPT 5.3 SIMPLE KEY AND VALUE EXTRACTION

```
>>> # what keys are available?
>>> dCars.keys()
['expensive','clean','fast']
>>> # what is the value associated with the key "clean"? >>> dCars.get("clean)"
'Prius'
```

Finally, you might want to iterate over a dictionary and extract and display all the key, value pairs.
CODE SCRIPT 5.4 ITERATING OVER A DICTIONARY

```
>>> for key, value in dCars.items():
... print key, value
expensive Rolls
clean Prius
fast Porsche
```

If you are not familiar with Python but have familiarity with other programming environments, Script 5.4 may seem a bit odd for several reasons.

1. We did not need to declare the variables' key and value, as Python takes care of that automatically. In this simple case both key and value are strings, but they are not required to be.

2. Since we know that dCars is a dictionary, the method items automatically returns both key and values. The keys are "clean," "expensive," and "fast" in this example and the values are the string names of the cars "Porsche," "Prius," and "Rolls."

Since Python will automatically handle the data typing for you, let us take a look at something a bit more complicated.
CODE SCRIPT 5.5 A MORE COMPLEX DICTIONARY EXAMPLE

```
>>> dTemp¼ {"2013:12:31":[22.0,"F","High"],
"2013:11:28":[24.5,"C","Low"], "2013:12:27":[32.7,"F","High"]} >>>
dTempItems ¼ dTemp.items()
>>> dTempItems.sort()
>>> for key, value in dTempItems:
.. . print key, value .. .
2013:11:28 [24.5,'C','Low'] 2013:12:27 [32.7,'F','High'] 2013:12:31
[22.0,'F','High']
```

As you can see in Code Script 5.5, the keys are strings that represent a date while value is actually a Python list that contains the time, the time reference, the temperature, and the units of the temperature.

Now that you have a good general understanding of the dictionaries, go create some of your own and experiment with the available methods. From this example, I hope you see the power of the Dictionary type and it has you thinking about other ways to apply this structure.

PIL test-before code

As with other modules and libraries, the best way to become comfortable with the library is to experiment with it. Remember our adage test-then code, this is especially true for PIL, mainly because the PIL uses a new Python language structure, the Dictionary data structure. In addition, the EXIF module of the PIL requires a solid understanding before you can put it to work for you. I will use the Python Shell and short scripts to demonstrate.

Determining the available EXIF TAGS

The PIL has two important sets of TAGS that provide keys that are used to access dictionary elements. They are EXIFTAGS and GPSTAGS. I developed a script to determine which EXIF TAGS are available by using the following method:

(1) Import the EXIFTAGS from the PIL library.
(2) Extract the dictionary items (the key, value pairs) from TAGS. (3) Sort them to make it easier later to identify them. (4) Print the sorted dictionary of key, value pairs.

CODE SCRIPT 5.6 EXTRACTING EXIF TAGS

```
>>> from PIL.EXIFTags import TAGS >>> EXIFTAGS ¼ TAGS.items()
>>> EXIFTAGS.sort()
>>> print EXIFTags
```

This produces the following printed list of available tags. Note this is not the data, just the global list of EXIF TAGS:

```
[(256,'ImageWidth'), (257,'ImageLength'), (258,'BitsPerSample'), (259,
'Compression'), (262,'PhotometricInterpretation'), (270,
'ImageDescription'), (271,'Make'), (272,'Model'), (273,'StripOffsets'), (274,
'Orientation'), (277,'SamplesPerPixel'), (278,'RowsPerStrip'), (279,
'StripByteConunts'), (282,'XResolution'), (283,'YResolution'), (284,
'PlanarConfiguration'), (296,'ResolutionUnit'), (301,'TransferFunction'),
(305,'Software'), (306,'DateTime'), (315,'Artist'), (318,'WhitePoint'), (319,
'PrimaryChromaticities'), (513,'JpegIFOffset'), (514,'JpegIFByteCount'),
(529,'YCbCrCoefficients'), (530,'YCbCrSubSampling'), (531,
'YCbCrPositioning'), (532,'ReferenceBlackWhite'), (4096,
'RelatedImageFileFormat'), (4097,'RelatedImageWidth'), (33421,
'CFARepeatPatternDim'), (33422,'CFAPattern'), (33423,'BatteryLevel'),
(33432,'Copyright'), (33434,'ExposureTime'), (33437,'FNumber'), (34665,
'EXIFOffset'), (34675,'InterColorProfile'), (34850,'ExposureProgram'),
(34852,'SpectralSensitivity'), (34853,'GPSInfo'), (34855,
'ISOSpeedRatings'), (34856,'OECF'), (34857,'Interlace'), (34858,
'TimeZoneOffset'), (34859,'SelfTimerMode'), (36864,'EXIFVersion'), (36867,
'DateTimeOriginal'), (36868,'DateTimeDigitized'), (37121,
'ComponentsConfiguration'), (37122,'CompressedBitsPerPixel'), (37377,
'ShutterSpeedValue'), (37378,'ApertureValue'), (37379,'BrightnessValue'),
(37380,'ExposureBiasValue'), (37381,'MaxApertureValue'), (37382,
'SubjectDistance'), (37383,'MeteringMode'), (37384,'LightSource'), (37385,
'Flash'), (37386,'FocalLength'), (37387,'FlashEnergy'), (37388,
'SpatialFrequencyResponse'), (37389,'Noise'), (37393,'ImageNumber'),
(37394,'SecurityClassification'), (37395,'ImageHistory'), (37396,
'SubjectLocation'), (37397,'ExposureIndex'), (37398,'TIFF/EPStandardID'),
(37500,'MakerNote'), (37510,'UserComment'), (37520,'SubsecTime'), (37521,
'SubsecTimeOriginal'), (37522,'SubsecTimeDigitized'), (40960,
'FlashPixVersion'), (40961,'ColorSpace'), (40962,'EXIFImageWidth'), (40963,
'EXIFImageHeight'), (40964,'RelatedSoundFile'), (40965,
'EXIFInteroperabilityOffset'), (41483,'FlashEnergy'), (41484,
'SpatialFrequencyResponse'), (41486,'FocalPlaneXResolution'), (41487,
'FocalPlaneYResolution'), (41488,'FocalPlaneResolutionUnit'), (41492,
```

'SubjectLocation'), (41493,'ExposureIndex'), (41495,'SensingMethod'), (41728,'FileSource'), (41729,'SceneType'), (41730,'CFAPattern')]

Time is a concept that allows society to function in an orderly fashion where all parties are able to easily understand the representation of time and agree on the sequence of events. Since 1972, the International standard for time is Coordinated Universal Time (UTC). UTC forms the basis for the official time source in most nations around the globe; it is governed by a diplomatic treaty adopted by the world community that designates National Metrology Institutes (NMIs) as UTC time sources. The National Institute of Standards and Technology in the United States and the National Physical Laboratory (NPL) in England are two examples of NMIs. There are about 50 similar metrology centers that are responsible for official time throughout the world. In addition, time zones can play a key role when examining digital evidence; they provide localization of events and actions based upon the computer's time zone setting. For example, if the computer's time zone is set to New York, the local time is À5 hours from UTC (or 5 hours behind). Finally, many U.S. States and foreign countries observe day light savings time, most modern operating systems automatically adjust for this observance which causes additional differences and calculations between UTC or when attempting to synchronize events across time zones.

Within the printed list, I have highlighted TAGS that are of immediate interest including GPSInfo, TimeZoneOffset, and DateTimeOriginal.

It is important to note that it is not guaranteed that all of these TAGS will be available for every image; rather this is the broad set of key value pairs that are potentially within an image. Each manufacturer independently determines which values will be included. Thus you must process the key value pairs for each target image to determine which TAGS are in fact available. I will show you how this is done shortly.

As you scan the list, you may find other TAGS that are of interest in your investigative activities and you can utilize the same approach defined here to experiment with those values.

Determining the available EXIF GPSTAGS

Next, I want to drill a bit deeper into the GPSInfo TAG, as our objective in this chapter is to extract GPS-based location data of specific images. I take a similar

approach to determine what GPSTAGS are available at the next level by writing the following lines of code.

CODE SCRIPT 5.7 EXTRACTING GPS TAGS

```
>>> from PIL.EXIFTags import GPSTAGS >>> gps ¼ GPSTAGS.items()
>>> gps.sort()
>>> print gps

[(0,'GPSVersionID'), (1,'GPSLatitudeRef'), (2,'GPSLatitude'),
(3,'GPSLongitudeRef'), (4,'GPSLongitude'), (5,'GPSAltitudeRef'),
(6,'GPSAltitude'),(7,'GPSTimeStamp'), (8,'GPSSatellites'), (9,'GPSStatus'),
(10,'GPSMeasureMode'), (11,'GPSDOP'), (12,'GPSSpeedRef'), (13,'GPSSpeed'),
(14,'GPSTrackRef'), (15,'GPSTrack'), (16,'GPSImgDirectionRef'), (17,
'GPSImgDirection'), (18,'GPSMapDatum'), (19,'GPSDestLatitudeRef'), (20,
'GPSDestLatitude'), (21,'GPSDestLongitudeRef'), (22,'GPSDestLongitude'),
(23,'GPSDestBearingRef'), (24,'GPSDestBearing'), (25,
'GPSDestDistanceRef'), (26,'GPSDestDistance')]
```

I have highlighted a few GPSTAGS that we need in order to pinpoint the location and time of the photograph. Again pointing out that there is no guarantee that any or all of these tags will be available within the subject image.

Longitude and latitude locations provide the ability to pinpoint a specific location on the earth. The Global Position System and GPS receivers built into smart phones and cameras can deliver very accurate geolocation information in the form of longitude and latitude coordinates. If so configured, this geolocation information can be directly embedded in photographs, movies, and other digital objects when certain events occur. Extracting this geolocation information from these digital objects and then entering that data into online services such as Mapquest, Google Maps, Google earth, etc., can provide evidence of where and when that digital object was created.

If we plan to utilize the TAGS to pinpoint the location, we need a better definition and understanding of the TAGS of interest, their meaning, and usage. GPSLatitudeRef: Latitude is specified by either the North or South position in degrees from the equator. Those points that are north of the equator range from 0 to 90 degrees. Those points that are south of the equator range from 0 to À90 degrees. Thus this reference value is used to define where the latitude is in

relationship to the equator. Since the value latitude is specified in EXIF as a positive integer, if the Latitude Reference is “S” or South, the integer value must be converted to a negative value.
GPSLatitude: Indicates the latitude measured in degrees. The EXIF data specify these as whole degrees, followed by minutes and then followed by seconds.
GPSLongitudeRef: Longitude is specified by either the east or west position in degrees. The convention is based on the Prime Meridian which runs through the Royal Observatory in Greenwich, England, which is defined as longitude zero degrees. Those points that are west of the Prime Meridian range from 0 to À180 degrees. Those points that are east of the Prime Meridian range from 0 to 180 degrees. Thus this reference value is used to define where the longitude value is in relationship to Greenwich. Since the value of longitude is specified in EXIF as a positive integer, if the Longitude Reference is west or “W,” the integer value must be converted to a negative value.
GPSLongitude: Indicates the longitude measured in degrees. The EXIF data specify these as whole degrees, followed by minutes and then followed by seconds.
GPSTimeStamp: Specifies the time as Universal Coordinated Time that the photo was taken.

Since the data contained within the EXIF structure are digital values, they are subject to manipulation and forgery. Thus, securing the images or obtaining them directly from the Smart Mobile device makes fraudulent modification less likely and much more difficult to accomplish.

Now that we have a basic understanding of what TAGS are potentially available within a photograph, let us examine an actual photograph. Figure 5.4 depicts an image downloaded from the Internet and will be the subject of this experiment.

FIGURE 5.4 Internet Photo Cat.jpg.

CODE SCRIPT 5.8 EXTRACT EXIF TAGS OF A SAMPLE IMAGE

```
>>> from PIL import Image
>>> from PIL.EXIFTags import TAGS, GPSTAGS
>>>
>>> pilImage ¼ Image.open("c:\\pictures\\cat.jpg)"
.. . EXIFData ¼ pilImage._getEXIF()
.. .
>>> catEXIF ¼ EXIFData.items()
>>> catEXIF.sort()
>>> print catEXIF
```

This script imports the PIL Image module that allows me to open the specific image for processing and to create the object pilImage. I can then use the object pilImage to acquire the EXIF data by using the ._getEXIF() method. As we did when extracting the EXIFTAGS and GPSTAGS, in this example I am extracting the dictionary items and then sorting them. Note, in this example I am extracting the actual EXIF data (or potential evidence) not the TAG references.

The output from the Code Script 5.3 is shown here.

[(271, ’Canon’), (272, ’Canon EOS 400D DIGITAL’), (282, (300, 1)), (283, (300, 1)), (296, 2), (306, ’2008:08:05 23:48:04’), (315, ’unknown’), (33432,

’Eirik Solheim www.eirikso. com’), (33434, (1, 100)), (33437, (22, 10)), (34665, 240), (34850, 2), (34853, {0: (2, 2, 0, 0), 1: ’N’, 2: ((59, 1), (55, 1), (37417, 1285)), 3: ’E’, 4: ((10, 1), (41, 1), (55324, 1253)), 5: 0, 6: (81, 1)}), (34855, 400), (36864, ’0221’), (36867, ’2008:08:05 20:59:32’), (36868, ’2008:08:05 20:59:32’), (37378, (2275007, 1000000)), (37381, (96875, 100000)), (37383, 1), (37385, 16), (37386, (50, 1)), (41486, (3888000, 877)), (41487, (2592000, 582)), (41488, 2), (41985, 0), (41986, 0), (41987, 0), (41990, 0)]

I have highlighted the GPSInfo data associated with this photograph. I was able to identify the GPSInfo TAG based on the output of the previous Code Script 5.1, whereby the output showed that the GPSInfo TAG had a key value of 34853. Therefore, this camera, the Canon EOS 400D DIGITAL, provides the GPS data highlighted between the curly braces. Closer examination of the GPS data reveals the specific key value pairs that are available. As you can see, the key value pairs embedded in the GPSInfo data dictionary are 0, 1, 2, 3, 4, 5, and 6. These TAGS refer to the standard GPS TAGS. In Code Script 5.2, I printed out all the possible GPS Tags, by referring to that output we discover the following GPS TAGS that were provided by the Cannon EOS 400D. They include:

0 GPSVersionID
1 GPSLatitudeRef
2 GPSLatitude
3 GPSLongitudeRef
4 GPSLongitude
5 GPSAltitudeRef
6 GPSAltitude

With this information I can calculate the longitude and latitude provided by the EXIF data within the photograph using the following process. But, first, I need to extract the latitude from the key value pair GPSTag 2:

((59, 1), (55, 1), (37417, 1285))
Each latitude and longitude value contains three sets of values. They are:

Degrees: (59, 1)
Minutes: (55, 1)
Seconds: (37,417, 1285)

You might be wondering why they are represented like this. The answer is the EXIF data does not hold floating point numbers (only whole integers). In order to provide more precise latitude and longitude data, each pair represents a ratio.

Next, I need to perform the ratio calculation to obtain a precise factional value, yielding:

Degrees: 59/1 ¼59.0
Minutes: 55/1¼55.0
Seconds: 37,417/1285¼29.1182879
Or more properly stated 59 degrees, 55 minutes, 29.1182879 seconds.

Note : GPSTag 1:“N” the Latitude Reference key value pair is a north reference from the equator and does not require us to convert this to a negative value, if it were “S” we would have make this additional conversion.

Next, most online mapping programs (i.e., Google Maps) require data to be provided as a fractional value. This is accomplished by using the following formula.

Degrees þ(minutes/60.0)þ(seconds/3600.0)
Note: the denominators of 60.0 and 3600.0 represent the number minutes (60.0) and seconds (3600.0) in each hour, respectively.

Thus this latitude value represented as floating point number would be:
Latitude¼39.0þ(55/60.0)þ(29.1182879/3600.0)
Latitude¼39.0þ.91666667þ.00808841¼ 59.9247551
Performing the same calculation on the longitude value extracted from GPSTag 4:

((10, 1), (41, 1), (55324, 1253))
With a GPSTag 3: “E”
Yields a longitude value of 10.6955981201
Giving us a latitude/longitude value of 59.924755, 10.6955981201

Using Google Maps, I can now plot the 59.924755, 10.6955981201 yielding a point just north and west of Oslo, Norway as shown in Figure 5.5. Noting that Oslo, Norway is in fact East of the Prime Meridian in Greenwich, England.

One final note, this camera did not include other GPSTAGS such as the

GPSTimestamp, which could have given us the time as reported by GPS.

p-ImageEvidenceExtractor fundamental requirements
Now that I have experimented with the PIL, discovered how to extract the EXIF

TAGS, the GPS tags, and the EXIF data from a target image, I want to define our first full extraction application.

Design considerations
Now that I have defined the basic approach, Table 5.1 and Table 5.2 define the program requirements and design considerations for the p-ImageEvidenceExtractor.

Next, I want to define the overall design of the p-gpsExtractor as depicted in Figure 5.6. I will again be using the command line parser to convey selections from the user. I will be directing output to standard out, a CSV file, and the forensic log. The user will specify a folder containing images to process. I have also created a class to handle forensic logging operations that I can reuse or expand in the future. Finally, I will be reusing a slightly modified version of the CVSWriter class I created in Chapter 3 for the pFish application.

CODE WALK-THROUGH

In order to get a feel for the program take a look at Figure 5.7, which is a screenshot of the WingIDE project. You notice on the far right this program contains five Python source files:

FIGURE 5.5
Map of GPS coordinates extracted From Cat.jpg.
Table 5.1 Basic Requirements for GPS/EXIF Extraction Requirement
Requirement number name

GPS-001 Command Line Arguments
GPS-002 Log GPS-003 Output

GPS-004 Error Handling Short description

Allow the user to specify a directory that contains sample images to attempt to extract GPS information from
The program must produce a forensic audit log The program should produce a list of latitude and longitude values from the GPS information found. The format must be simple lat, lon values in order to be pasted into online mapping applications
The application must support error handling and logging of all operations performed. This will include a textual description and a timestamp

Table 5.2 Design Consideration and Library Mapping Requirement Design considerations Library selection

User Input (001) User input for keyword and target file
Output (003) The extracted GPS location will be printed out using simple built-in print commands
(Additional)

(002 and 004) Logging and Error Handling Extract additional EXIF and GPS values and record the data for each file encountered in a CSV file Errors may occur during the
processing of the files and EXIF data. Therefore, handling these potential errors must be rigorous
I will be using argparse from the Standard Library module to obtain input from the user
I will be using the Python Image Library and the
standard Python language to extract and format the output in order to cut and paste directly into the Web page: http://www.
darrinward.com/lat-long/ Use the Python Image
Library and Python CVS Library
The Python Standard
Library includes a logging facility which I can leverage to report any events or errors that occur during processing. Note for this application I will create a reuseable logging class

user Standard out
Program Arguments

p-gpsExtractor CSV
results

Folder with

file

images
Event and error log

p-gpsExtractor context diagram FIGURE 5.6
p-gpsExtractor context diagram.

FIGURE 5.7
WingIDE Project Overview.

Main Program
p-gpsExtractor.py: This is the Main Program for this application handling the overall program flow and the processing of each potential image file contained in a folder specified by the user via the command line. Main writes data to standard out in order to make it easy to cut and paste that data into a Web page for map viewing. Main also handles output to the CSV file that provides the results for each of the files processed, as well as any forensic log entries.

Class Logging
classLogging.py: The new class logger abstracts the logging function in a Python class for handling Forensic Logging functions. This will allow for log object passing for future applications to any module or function eliminating any need for global variables.

cvsHandler
csvHandler.py: This modified class for handling CSV File Creation and Writing provides a single point interface to writing output files in CVS format, once again to abstract the functions and make interface simpler. More work will be done on this class in the future to abstract this further.

Command line parser
commandParser.py: I separated out the commandParser.py into its own file for greater portability for this application and for future applications. This allows command line parser to be handled, modified, or enhanced and is more loosely coupled from the Main Program. Like the Logging functions this will eventually be turned into a Class-based module to allow cross-module access and interface.

EXIF and GPS Handler
_modEXIF.py: This module directly interfaces with the PIL to perform the needed operations to extract both EXIF and GPS data from image files that contain EXIF data.

Examining the code Main Program

```
# GPS Extraction # Python-Forensics # No HASP required #
import os
import _modEXIF
import _csvHandler
import _commandParser
from classLogging import ForensicLog
```

For the main program we need to import os to handle folder processing, the command line processor, the cvsHandler and the new ForensicLog Class. # Process the Command Line Arguments
userArgs ¼ _commandParser.ParseCommandLine()

As I typically do, I process the command line arguments to obtain the user specified values. If command parsing is successful we create a new logging object to handle any necessary forensic log events and write the first log event.

```
# create a log object
logPath ¼ userArgs.logPath+"ForensicLog.txt"
oLog ¼ _ForensicLog(logPath)
```

oLog.writeLog("INFO", "Scan Started)" Next I create a CSV File Object that I will use to write data to the CSV results file that will contain the files processed and the resulting GPS data extracted csvPath ¼
userArgs.csvPath+"imageResults.csv" oCSV ¼
_csvHandler._CSVWriter(csvPath)

Needing to obtain the user specified target directory for our GPS extraction, I attempt to open and then process each of the files in the target directory. If any errors occur they are captured using the try except model and appropriate errors are written to the forensic log file.

```
# define a directory to scan
scanDir ¼ userArgs.scanPath
try:
picts¼ os.listdir(scanDir)

except:
oLog.writeLog("ERROR", "Invalid Directory "+ scanDir) exit(0)

for aFile in picts: targetFile ¼ scanDir+aFile if os.path.isfile(targetFile):
```

Once the preliminaries are out of the way, I can now utilize the _modEXIF module to process each target file, with the result being a gpsDictionary object if successful. This object will contain the EXIF data that is available for this file as embedded in the EXIF record. Before using the data, I check to ensure that the gpsDictionary object contains data.

```
gpsDictionary ¼ _modEXIF.ExtractGPSDictionary(targetFile) if
(gpsDictionary):

# Obtain the Lat Lon values from the gpsDictionary # converted to degrees
# the return value is a dictionary key value pairs

dCoor ¼ _modEXIF.ExtractLatLon(gpsDictionary)
```

Next, I call another _modEXIF function to process further the gpsDictionary. This function as you will see, processes the GPS coordinates, and if successful returns the raw coordinate data. Then using the Dictionary methods you learned about earlier in this chapter, I extract the latitude and longitude data from the dictionary by using the get method.

```
lat ¼ dCoor.get("Lat)"
latRef ¼ dCoor.get("LatRef)"
lon ¼ dCoor.get("Lon)"
lonRef ¼ dCoor.get("LonRef)"
```

if (lat and lon and latRef and lonRef):

Upon successful extraction of the latitude and longitude values I print the results to standard out, write the data to the CSV file and update the forensic log accordingly. Or if unsuccessful, I report the appropriate data to the forensic log file.

print str(lat)+','+str(lon)

write one row to the output csv file
oCSV.writeCSVRow(targetFile, EXIFList[TS], EXIFList [MAKE], EXIFList[MODEL],latRef, lat, lonRef,
lon)

oLog.writeLog("INFO", "GPS Data Calculated for:" + targetFile)
else:
oLog.writeLog("WARNING", "No GPS EXIF Data for"+ targetFile)
else:
oLog.writeLog("WARNING", "No GPS EXIF Data for "+ targetFile)
else:
oLog.writeLog("WARNING", targetFile + " not a valid file)" Finally before program exit I delete the forensic log and CSV objects which in turn closes the log and the associated CSV file.
Clean up and Close Log and CSV File
del oLog
del oCSV

EXIF and GPS processing
The core data extraction elements of the p-gpsExtractor are contained within the _modEXIF.py module.

#
Data Extraction - Python-Forensics
Extract GPS Data from EXIF supported Images (jpg, tiff) # Support Module
#

I start by importing the libraries and modules needed by _modEXIF.py. Important module of note is the Python Imaging Library and associated capabilities. import os # Standard Library OS functions
from classLogging import _ForensicLog # Logging Class From the PIL I import

the Image Library and the EXIF and GPS Tags in order to index into the dictionary structures, much like I did in the script examples 5-6 thru 5-8.
# import the Python Image Library # along with TAGS and GPS related TAGS
from PIL import Image
from PIL.EXIFTags import TAGS, GPSTAGS

Now let’s take a look and breakdown the individual functions. I will start with the extraction of the GPS Dictionary. The function take the full path name of the target file that the extraction is to occur with.

#
Extract EXIF Data
#
# Input: Full Pathname of the target image #
# Return: gpsDictionary and extracted EXIFData list #
def ExtractGPSDictionary(fileName):

Using the familiar try / except module I attempt to open the filename provided and if successful I then extract the EXIF data by using the PIL getEXIF() function.

try: pilImage ¼ Image.open(fileName) EXIFData ¼ pilImage._getEXIF()

except Exception:
If exception occurs from PIL processing # Report the
return None, None

Iterate through the EXIFData # Searching for GPS Tags

Next, I’m going to iterate through the EXIFData TAGS and collect some basic EXIF data and then obtain the GPSTags if they are included within the EXIF data of the specific image.

imageTimeStamp ¼ "NA" CameraModel ¼ "NA" CameraMake ¼ "NA"

if EXIFData: for tag, theValue in EXIFData.items(): # obtain the tag
tagValue ¼ TAGS.get(tag, tag) # Collect basic image data if available

As I iterate through the tags that are present within this image, I check for the presence of the DataTimeOriginal along with the camera make and model in

order to include them in the result file.

if tagValue ¼¼'DateTimeOriginal':
imageTimeStamp ¼ EXIFData.get(tag)
if tagValue ¼¼ "Make":
cameraMake ¼ EXIFData.get(tag)
if tagValue ¼¼'Model':
cameraModel ¼ EXIFData.get(tag) I also check for the GPSInfo TAG, and if present I iterate through GPSInfo and extract the GPS Dictionary using the GPSTAGS.get()functions. # check the tag for GPS if tagValue ¼¼ "GPSInfo": # Found it !
Now create a Dictionary to hold the GPS Data

gpsDictionary ¼ {}
Loop through the GPS Information

for curTag in theValue:
gpsTag¼ GPSTAGS.get(curTag, curTag) gpsDictionary[gpsTag] ¼ theValue[curTag]

Next, I create a simple Python List to hold the EXIF Timestamp and Camera Make and Model. Then we return gpsDictionary and the basicEXIFData that was collected.

basicEXIFData ¼ [imageTimeStamp, cameraMake, cameraModel]
return gpsDictionary, basicEXIFData else:

If the target file doesn’t contain any EXIF data I return None in order to prevent any further processing of this file. Note None is a built in Python constant that is used to represent empty or missing data.

return None, None
End ExtractGPSDictionary ¼¼¼

Extract the Latitude and Longitude Values
from the gpsDictionary
#

The ExtractLatLon function takes as input a gps Dictionary structure. def ExtractLatLon(gps):
to perform the calculation we need at least
lat, lon, latRef and lonRef

Before I attempt the conversion, I need to validate the gps Dictionary contains the proper key value pairs. This includes the latitude, longitude, latitude reference and longitude reference.

if (gps.has_key("GPSLatitude)" and
gps.has_key("GPSLongitude)"
and gps.has_key("GPSLatitudeRef)"
and gps.has_key("GPSLatitudeRef)"):

Once we have the minimum inputs I extract the individual values.

latitude¼ gps["GPSLatitude"]
latitudeRef ¼ gps["GPSLatitudeRef"]
longitude ¼ gps["GPSLongitude"]
longitudeRef ¼ gps["GPSLongitudeRef"]

I then call the conversion function for both latitude and longitude values. lat ¼ ConvertToDegrees(latitude)
lon ¼ ConvertToDegrees(longitude)
Next I need to account for the latitude and longitude reference values and set the proper negative values if necessary.
Check Latitude Reference
If South of the Equator then lat value is negative if latitudeRef ¼¼ "S":
lat ¼ 0 - lat

Check Longitude Reference
If West of the Prime Meridian in
Greenwich then the Longitude value is negative

if longitudeRef ¼¼ "W": lon ¼ 0 - lon

Once this has been accounted for, I create a new dictionary to hold the final values and return that to the caller. In this case the main program.

```
gpsCoor ¼ {"Lat": lat, "LatRef":latitudeRef,
"Lon": lon, "LonRef": longitudeRef}
return gpsCoor else:
```

Once again, if the minimum values are not present, I return the built in Python constant None to indicate and empty return.

```
return None
```

The final support function is the ConvertToDegrees function that converts the GPS data into a floating point value. This code is very straightforward and follows the formula presented in the narrative of this chapter. The only important aspect to point out is the try/except model used whenever dividing two numbers, as a divide by zero would cause a failure to occur and the program to abort. As with any data coming from the wild, anything is possible and we need to account for possible zero division. I have seen this with EXIF data, when the value for seconds is zero the ratio is reported as 0:0. Since this is a legal value, I simply set the degrees, minutes, or seconds to zero when the exception occurs.

```
#
# Convert GPSCoordinates to Degrees
#
# Input gpsCoordinates value from in EXIF Format
#

def ConvertToDegrees(gpsCoordinate):
d0 ¼ gpsCoordinate[0][0]
d1 ¼ gpsCoordinate[0][1]

try:
degrees ¼ float(d0) / float(d1)
except:
degrees ¼ 0.0

m0 ¼ gpsCoordinate[1][0] m1 ¼ gpsCoordinate[1][1] try:

minutes ¼ float(m0) / float(m1) except:
minutes¼0.0
s0 ¼ gpsCoordinate[2][0] s1 ¼ gpsCoordinate[2][1] try:
```

seconds ¼ float(s0) / float(s1) except:
seconds ¼ 0.0

floatCoordinate ¼ float (degrees + (minutes / 60.0) + (seconds / 3600.0))
return floatCoordinate

Logging Class
For this application I have include a new class logging that abstracts and simplifies the handling of forensic logging events across modules.

import logging
#
Class: _ForensicLog
#
Desc: Handles Forensic Logging Operations
#
Methods
constructor: # writeLog:
destructor:

class _ForensicLog: Initializes the Logger
Writes a record to the log Writes an information message

and shuts down the logger

init is the constructor method that is called whenever a new ForensicLog object is created. The method initializes a new Python logging object using the Python Standard Library. The filename of the log is passed in as part of the object instantiation. If an exception is encountered a message is sent to standard output and the program is aborted.

def __init__(self, logName):
try:
# Turn on Logging
logging.basicConfig(filename¼logName,

level¼logging.DEBUG,format¼'%(asctime) s %(message)s') except:
print "Forensic Log Initialization
Failure ... Aborting"

exit(0)

The next method is writeLog which takes two parameters (along with self), the first is the type or level of the log event (INFO, ERROR or WARNING) and then a string representing the message that will be written to the log.

def writeLog(self, logType, logMessage): if logType ¼¼ "INFO":
logging.info(logMessage)

elif logType ¼¼ "ERROR":
logging.error(logMessage)
elif logType ¼¼ "WARNING": logging.warning(logMessage)
else:
logging.error(logMessage)
return

Finally the del method is used to shutdown logging and to close the log file. The message Logging Shutdown is sent to the log as an information message just prior to shutdown of the log.

def __del__(self):
logging.info("Logging Shutdown)"
logging.shutdown()

Command line parser
Command line parser handles the user input, provides validation for the arguments passed, and reports error along with help text to assist the user.

import argparse # Python Standard Library - Parser for command-line options, arguments
import os # Standard Library OS functions
# Name: ParseCommand() Function
#
Desc: Process and Validate the command line arguments
use Python Standard Library module argparse
#
Input: none
#
Actions:

#
#
Uses the standard library argparse
to process the command line

#
def ParseCommandLine():

gpsExtractor requires three inputs 1) the directory path for the forensic log, 2) the directory path to be scanned and 3) the directory path for the CSV result file. All these directory paths must exist and be writable in order for parsing to succeed.

parser ¼ argparse.ArgumentParser('Python gpsExtractor) parser.add_argument('-v','--verbose', help¼"enables printing of additional program messages", action¼'store_true') parser.add_argument('-l','--logPath', type¼ ValidateDirectory, required¼True, help¼"specify the directory for forensic log output file")

parser.add_argument('-c','--csvPath', type¼ ValidateDirectory, required¼True, help¼"specify the output directory for the csv file)”

parser.add_argument('-d','-scanPath', type¼ValidateDirectory, required¼True, help¼"specify the directory to scan)"

theArgs ¼ parser.parse_args() return theArgs
End Parse Command Line ¼¼¼

if os.access(theDir, os.W_OK):
return theDir
else:
raise argparse.ArgumentTypeError
('Directory is not writable')
#End ValidateDirectory
¼¼¼

'Lat Ref','Latitude', 'Lon Ref','Longitude'))

except:
log.error('CSV File Failure')

The actual writeCSVRow method takes as input the individual columns defined and writes them to the file. In addition, the function converts the floating point latitude and longitude values to properly formatted strings.

def writeCSVRow(self, fileName, timeStamp, CameraMake, CameraModel,latRef, latValue, lonRef, lonValue):

latStr¼'%.8f'% latValue
lonStr¼'%.8f'% lonValue

self.writer.writerow((fileName, timeStamp, CameraMake, CameraModel, latRef, latStr, lonRef, lonStr))

Finally the del method closes the csvFile for proper deconstruction of the object.
def __del__(self):
self.csvFile.close()
Full code listings

#
GPS Extraction
Python-Forensics # No HASP required

import os
import _modEXIF
import _csvHandler
import _commandParser
from classLogging import _ForensicLog

Offsets into the return EXIFData for # TimeStamp, Camera Make and Model

TS ¼ 0
MAKE ¼ 1
MODEL ¼ 2

```
# Process the Command Line Arguments
userArgs ¼ _commandParser.ParseCommandLine()

# create a log object
logPath ¼ userArgs.logPath+"ForensicLog.txt" oLog ¼ _ForensicLog(logPath)

oLog.writeLog("INFO", "Scan Started") csvPath ¼
userArgs.csvPath+"imageResults.csv" oCSV ¼
_csvHandler._CSVWriter(csvPath)

# define a directory to scan scanDir ¼ userArgs.scanPath try:

picts ¼ os.listdir(scanDir)
except:
oLog.writeLog("ERROR", "Invalid Directory "+ scanDir)
exit(0)

print "Program Start" print

for aFile in picts : targetFile ¼ scanDir+aFile if os.path.isfile(targetFile):

gpsDictionary, EXIFList ¼ _modEXIF.ExtractGPSDictionary (targetFile)
if (gpsDictionary):

# Obtain the Lat Lon values from the gpsDictionary # Converted to degrees
# The return value is a dictionary key value pairs

dCoor¼ _modEXIF.ExtractLatLon(gpsDictionary) lat ¼ dCoor.get("Lat") latRef
¼ dCoor.get("LatRef") lon ¼ dCoor.get("Lon")
lonRef ¼ dCoor.get("LonRef")
if ( lat and lon and latRef and lonRef): print str(lat)+','+str(lon)

# write one row to the output file
oCSV.writeCSVRow(targetFile, EXIFList[TS], EXIFList [MAKE],
EXIFList[MODEL],latRef, lat, lonRef, lon) oLog.writeLog("INFO", "GPS Data
Calculated for :" + targetFile)

else :
oLog.writeLog("WARNING", "No GPS EXIF Data for "+ targetFile)
```

```
else :
oLog.writeLog("WARNING", "No GPS EXIF Data for "+ targetFile)

else:
oLog.writeLog("WARNING", targetFile + " not a valid file")

# Clean up and Close Log and CSV File del oLog
del oCSV
import argparse
for command-line options, arguments import os
# Python Standard Library - Parser

# Standard Library OS functions

# Name: ParseCommand() Function #
# Desc: Process and Validate the command line arguments
# use Python Standard Library module argparse
#
# Input: none
#
# Actions:
# Uses the standard library argparse to process the command

line
#
def ParseCommandLine():

parser¼ argparse.ArgumentParser('Python gpsExtractor')

parser .add_argument('- v','--verbose', help¼"enables printing of additional
program messages", action¼'store_true')
parser.add_argument('- l','--logPath', type¼ ValidateDirectory, required¼True,
help¼"specify the directory for forensic log output file")
parser.add_argument('- c ','--csvPath', type¼ ValidateDirectory, required¼True,
help¼"specify the output directory for the csv file")
parser.add_argument('- d','--scanPath', type¼ ValidateDirectory, required¼True,
help¼"specify the directory to scan")

theArgs ¼ parser.parse_args() return theArgs
```

End Parse Command Line ¼¼¼

```
def __del__(self):
logging.info("Logging Shutdown")
logging.shutdown()

#
# Data Extraction - Python-Forensics
# Extract GPS Data from EXIF supported Images (jpg, tiff)
# Support Module
#
import os # Standard Library OS functions from classLogging import
_ForensicLog # Abstracted Forensic Logging

Class
# import the Python Image Library # along with TAGS and GPS related TAGS
from PIL import Image
from PIL.EXIFTags import TAGS, GPSTAGS

#
# Extract EXIF Data
#
# Input: Full Pathname of the target image #
# Return: gps Dictionary and selected EXIFData list #

def ExtractGPSDictionary(fileName):

try :
pilImage ¼ Image.open(fileName) EXIFData ¼ pilImage._getEXIF()

except Exception :
# If exception occurs from PIL processing # Report the
return None, None

# Iterate through the EXIFData
# Searching for GPS Tags

imageTimeStamp ¼ "NA"
CameraModel ¼ "NA"
CameraMake ¼ "NA"

if EXIFData: for tag, theValue in EXIFData.items(): # obtain the tag
```

tagValue ¼ TAGS.get(tag, tag) # Collect basic image data if available if tagValue ¼¼'DateTimeOriginal': imageTimeStamp ¼ EXIFData.get(tag) if tagValue ¼¼ "Make":
cameraMake ¼ EXIFData.get(tag) if tagValue ¼¼'Model':
cameraModel ¼ EXIFData.get(tag) # check the tag for GPS if tagValue ¼¼ "GPSInfo": # Found it !
Now create a Dictionary to hold the GPS Data gpsDictionary ¼ {}

# Loop through the GPS Information
for curTag in theValue:
gpsTag ¼ GPSTAGS.get(curTag, curTag)
gpsDictionary[gpsTag] ¼ theValue[curTag]

basicEXIFData ¼ [imageTimeStamp, cameraMake, cameraModel]
return gpsDictionary, basicEXIFData

else:
return None, None
End ExtractGPSDictionary
¼¼¼

¼ gps["GPSLongitude"]

longitudeRef ¼ gps["GPSLongitudeRef"]

lat ¼ ConvertToDegrees(latitude) lon ¼ ConvertToDegrees(longitude) # Check Latitude Reference
If South of the Equator then lat value is negative

if latitudeRef ¼¼ "S": lat ¼ 0 lat

Check Longitude Reference
If West of the Prime Meridian in
Greenwich then the Longitude value is negative

if longitudeRef ¼¼ "W": lon ¼ 0- lon
gpsCoor ¼ {"Lat": lat, "LatRef":latitudeRef, "Lon": lon, "LonRef": longitudeRef}
return gpsCoor else:
return None
End Extract Lat Lon
¼¼¼

s0 ¼ gpsCoordinate[2][0] s1 ¼ gpsCoordinate[2][1] try:

seconds ¼ float(s0)/ float(s1) except:
seconds ¼ 0.0

floatCoordinate ¼ float (degrees+(minutes / 60.0)+(seconds / 3600.0))
return floatCoordinate
import csv #Python Standard Library - reader and writer for csv files #
# Class: _CSVWriter
#
Desc: Handles all methods related to comma separated value operations
Methods constructor: Initializes the CSV File
writeCVSRow: # writerClose: Writes a single row to the csv file

Closes the CSV File
class _CSVWriter: def __init__(self, fileName):

try :
# create a writer object and then write the header row self.csvFile ¼
open(fileName, 'w b')
self.writer¼ csv.writer(self.csvFile, delimiter¼',', quoting¼csv.QUOTE_ALL)
self.writer.writerow(('Image Path','M a k e','M o d e l','UTC Time',

'L a t R e f','Latitude','L o n R e f','Longitude','A l t R e f','Altitude')) except:
log.error('CSV File Failure')

def writeCSVRow(self, fileName, cameraMake, cameraModel, utc, latRef,
latValue, lonRef, lonValue, altRef, altValue): latStr¼'%.8f'% latValue
lonStr¼'%.8f'% lonValue
altStr¼'%.8f'% altValue
self.writer.writerow(fileName, cameraMake, cameraModel, utc,
latRef, latStr, lonRef, lonStr, altRef, AltStr)

def __del__(self):
self.csvFile.close()

Program execution
The p-gpsExtractor produces three separate results: (1) The standard output rendering of the latitude and longitude values extracted from the target files. (2)

The CSV file that contains the details of each EXIF and GPS extraction. (3) The forensic log file that contains the log of program execution. Execution of the program is accomplished by using the following command and line options.

C:\pictures\Program>Python p-GPSExtractor.py-d c:\pictures\ -c c:\pictures\Results\ -l c:\pictures\Log\

Figure 5.8 depicts the results of program execution. I have selected the output results and copied them into the paste buffer in order to plot all the points within an online mapping program.

Figure 5.8 contains a screenshot of a Windows execution of p-gpsExtractor.
Summary Questions 159

FIGURE 5.8
p-gpsExtractor.py execution.

In Figure 5.9, I paste the selected coordinates from the program execution into the online Web site, http://www.darrinward.com/lat-long/ [Ward] and submit the values. The page plots each of the points on the map as a push pin. Figures 5.10 and 5.11 illustrate the zoom into the locations in Germany.

In addition to the mapped data, I also generated a .csv result file for the scan. Figure 5.12 shows the resulting .csv file.
Finally, I also create a Forensic Log file associated with program execution. This file contains the forensic log results shown in Figure 5.13.

CHAPTER REVIEW

In this chapter, I put to use new Python language elements including the

dictionary in order to handle more complex data types and to interface with the PIL. I also utilized the PIL to systematically extract EXIF data from photographs including GPS data when available with photographs. I demonstrated not only how to extract the GPS raw data but also how to calculate the lon and lat positions and then later integrated the output of the program into an online mapping program to plot the locations of the photographs. In addition, I demonstrated how to extract data from the EXIF data structure and included some of those values in a resulting .csv file that contains information about the file, camera, timestamp, and location information.

SUMMARY QUESTIONS

1. Based on the experimentation with the Python built-in data type dictionary, what limitations for key and value do you see?
2. Choose five additional fields from the EXIF structure, develop the additional code to extract them, locate several photographs that contain this additional data, and add them to the .csv output.

FIGURE 5.9
Mapping the coordinates extracted from photos.

FIGURE 5.10
Map zoom into Western Europe.

FIGURE 5.11
Map zoom to street level in Germany.

	Image Path	TimeStamp	Camera Make	Camera Model	Lat Ref	Latitude	Lon Ref	Longitude
2	c:\pictures\Biking.jpg	2006:02:11 11:06:37	Canon	Canon PowerShot A80	N	33.87546082	W	-116.3016196
3	c:\pictures\Castle.JPG	2012:06:09 12:42:24	PENTAX	PENTAX K-5		55.00733833		11.91093333
4	c:\pictures\Cat.jpg	2008:08:05 20:59:32	Canon	Canon EOS 400D DIGITAL	N	59.92475508	E	10.69559812
5	c:\pictures\Deutchland.JPG	2010:06:23 15:32:25	Apple	iPhone 3G	N	47.975	E	7.82966667
6	c:\pictures\Disney.jpg	2010:08:18 11:38:37	Canon	Canon EOS 1000D	N	28.41878951	W	-81.58097209
7	c:\pictures\dscn0011.jpg	2009:03:14 13:46:34	NIKON	COOLPIX P6000	N	42.501235	W	-83.25066333
8	c:\pictures\kinderscout.jpg	2008:01:12 12:06:52	NIKON CORPORATION	NIKON D50	N	53.40716667	W	-1.90366667
9	c:\pictures\Munich.JPG	2010:06:21 16:00:57	Apple	iPhone 3G	N	48.14133333	E	11.57666667
10	c:\pictures\Turtle.jpg	2008:05:08 16:55:58	Canon	Canon EOS 5D	N	25.3384	E	34.739666

FIGURE 5.12

Snapshot of Results.csv file.
Additional Resources 163

FIGURE 5.13
Snapshot of the Forensic Log file.

3. Locate other online mapping resources (for example, ones that would allow you to tag the photo, for example, with a filename) and then use the mapping function to render the additional data.

4. Using the PIL, develop a program that will maliciously modify the latitude, longitude, and/or timestamp values of an existing photograph.
5. Expand the extraction method to include altitude and then find a mapping program that would allow you to map latitude, longitude, and altitude. Then locate photos with varying altitudes, for example, photos taken from hot air balloons.

Additional Resources

FCC 911 Wireless Service Guide. http://www.fcc.gov/guides/wireless-911-services. Python Imaging Library. https://pypi.python.org/pypi/PIL.
The Python Standard Library. http://docs.python.org/2/library/.
Web Based multiple coordinate mapping website.
http://www.darrinward.com/lat-long/.

CHAPTER

Forensic Time 6

CHAPTER CONTENTS

Introduction 165
Adding Time to the Equation 167
The Time Module 169
The Network Time Protocol 173
Obtaining and Installing the NTP Library ntplib 174
World NTP Servers 177
NTP Client Setup Script 179
Chapter Review 181
Summary Questions 181
Additional Resources 181

INTRODUCTION

Before we can even consider time from an evidentiary point of view, we need to better understand what time is. I will start with an excerpt from one of my favorite books “Longitude” by Dava Sobel:

“Anyone alive in the eighteenth century would have known that the longitude problem was the thorniest scientific dilemma of the day and had been for centuries. Lacking the ability to measure their longitude, sailors throughout the great ages of the exploration had been literally lost at sea as soon as they lost sight of land.

The quest for a solution had occupied scientists and their patrons for the better part of two centuries when, in 1714, England’s Parliament upped the ante by

offering a king's ransom (£20,000, approximately $12 million in today's currency) to anyone whose method or device proved successful. The scientific establishment throughout Europe from Galileo to Sir Isaac Newton had mapped the heavens in the both hemispheres in its certain pursuit of a celestial answer. In stark contrast, one man, John Harrison, dared to imagine a mechanical solution – a clock that would keep precise time at sea, something no clock had every been able to do on land. Mr. Harrison's invention resulted in a solution to the greatest scientific problem of this time." [Sobel, 1995]

The basis of Harrison's clock and solution still guide navigation today as the use of time within GPS and other navigation systems remains the key underlying

© 2014 Elsevier Inc. All rights reserved.

FIGURE 6.1
John Harrison H1 clock

component that allows us to calculate where we are. Figure 6.1 depicts the first Harrison sea clock H1.

Mr. Harrison's lifetime work was the impetus for the formation of Greenwich Mean Time or GMT. In today's high-speed global economy and communication infrastructure, precise, accurate, reliable, nonforgeable, and nonreputable time is

almost as elusive as it was in the eighteenth century. Of course, atomic, rubidium, and cesium clocks are guaranteed to deliver accurate time—and without missing a beat—for over a 1000 years. But, delivering that accuracy to digital evidence is elusive. My favorite saying is, “anyone can tell you what time it is, proving what time it was is the still the challenge” [Hosmer 2002].

Time is a concept that allows society to function in an orderly fashion where all parties are able to easily understand the representation of time and agree on a sequence of events. Since 1972, the international standard for time is Coordinated Universal Time or UTC. UTC forms the basis for the official time source in most nations around the globe; it is governed by a diplomatic treaty adopted by the world community that designates National Measurement (or Metrology) Institutes (NMIs) as UTC time sources. The National Institute of Standards and Technology (NIST) in the United States, the Communications Research Laboratory (CRL) in Japan, and the National Physical Laboratory (NPL) in the United Kingdom are examples of NMIs. There are about 50 similar metrology centers that are responsible for official time throughout the world.

Time has been a critical element in business for hundreds of years. The time—via a mechanical record—established and provided evidence as to the “when” of

business transactions and events. (Usually recorded on paper, these time records provided an evidentiary trail due to a number of unique attributes—the distinctiveness of the ink embedded in the paper, the style and method of type, whether mechanical or hand written, the paper and its characteristics, and the detectability of modifications, insertions, or deletions). In the electronic world where documents and timestamps are simply bits and bytes, authoritative proof of the time of an event has been hard to nail down. Since the security of time is absolutely linked to its origination from an official source, the reliance on easily manipulated time sources (e.g., typical computer system clocks) is problematic.

Precision and reliability have long been the major factors driving the design of electronic timing technology. The next technology wave adds the dimensions of authenticity, security, and auditability to digital timestamps. These three attributes, when applied to documents, transactions, or any other digital entity, provide the following advantages over traditional computer clock-based timestamps:

Assurance that the time came from an official source Assurance that time has not been manipulated An evidentiary trail for auditing and nonrepudiation

Authentication (knowledge of the originator's identity) and integrity (protection against content modification) are essential elements in trusted transactions. But if one cannot trust the time, can the transaction really be trusted? Secure timestamps add the missing link in the trusted transaction equation: "when" did it happen? However, for the timestamp to provide a reliable answer to that question, the time upon which it is based must be derived from a trusted source.

ADDING TIME TO THE EQUATION

Using the best practices afforded to us, today digital signatures are able to successfully bind "who" (the signer) with the "what" (the digital data). However, digital signatures have shortcomings that leave two critical questions unanswered:

When did the signing of the digital evidence occur?1.
2. How long can we prove the integrity of the digital evidence that we signed?

For both of these questions, time becomes a critical factor in proving the integrity of digital evidence. Through the work of the Internet Engineering Task Force (IETF) and private companies, timestamping has advanced to a realistic deployment stage.

To understand this, we must first understand a little about time itself and what is necessary if we choose to utilize time as a digital evidence trust mechanism. From ancient societies to the present day, time has been interpreted in many ways. Time is essentially an agreement that allows society to function in an orderly fashion—where all parties are easily able to understand the representation. Some examples of time measurement are included in Figure 6.2.

1967 1972 2009

1. Early calendars based on moon phases

2. Egyptians discover solar year
3. Atomic clocks utilized

4. UTC time
named official time

5. NIST and USNO official US time sources

FIGURE 6.2 A very brief history of time.

As shown in Figure 6.2, the first calendars known to man were based upon the moon because everyone could easily agree on this as a universal measure of time. Moving forward, the Egyptians were the first to understand the solar year and they were able to develop a calendar based on the rotation of the earth around the sun. In 1967, an international agreement defined the unit of time as the second, measured by the decay of Cesium using precision instruments known as atomic clocks. And in 1972, the Treaty of the Meter (also known as the Convention of the Metre, established in 1875) was expanded to include the current time reference known as UTC, which replaced GMT as the world's official time.

Today, more than 50 national laboratories operate over 300 atomic clocks in order to provide a consistent and accurate UTC. Thus, in order to create a trusted source of time, you must first reference an official time source. In the United States, we have two official sources of time; the National Institute of Standards and Technology (NIST) in Boulder, CO for commercial time and the United States Naval Observatory (USNO) for military time.

Despite the broad accessibility of accurate time (watches, computer clocks, time servers, etc.), the incorporation of trusted time within a system requires a secure, auditable digital date/timestamp. Unless you have direct connection to the NMIs, most trusted sources today utilize GPS as a source of time. When used properly to tune Rubidium timeservers, the accuracy provided by GPS signals provides the exactness and precision that are required for most transactions and record keeping. This is true if and only if these requirements, standards, and processes are followed:

Accuracy : The time presented is from an authoritative source and is accurate to the precision required by the transaction, whether day, hour, or millisecond.
Authentication: The source of time is authenticated to a trusted source of time, such as an NMI timing lab, GPS signal, or the NIST long-wave standard time

signal (WWVB in the United States).
Integrity: The time should be secure and not be subject to corruption during normal “handling.” If it is corrupted, either inadvertently or deliberately, the corruption should be apparent to a third party.
Nonrepudiation: An event or document should be bound to its time so that the association between event or document and the time cannot be later denied.
Accountability:The process of acquiring time, adding authentication and integrity, and binding it to the subject event should be accountable so that a third party can determine that due process was applied, and that no corruption transpired.

Most people reading this book will quickly recognize that from a practical perspective the timestamps that we collect during digital investigations rarely meet the standards set forth above. However, the knowledge of the standards we are striving for is vital. This should cause you to question everything—especially anything that is related to time. To get started, let us take a deep dive into the Python Standard Library examining the time, datetime, and calendar modules.

THE TIME MODULE

The time module has many attributes and methods that assist in processing timerelated information. To get started, I will use the proven technique of “test before code” to experiment with the time module.

The first issue we encounter is the definition of “the epoch.” An epoch is defined as a notable event that marks the beginning of a period in time. Modern digital computers reference specific points in history, for example, most Unix-based systems select midnight on January 1, 1970, whereas Windows uses January 1, 1601; and Macintosh systems typically reference January 1, 1904.

The original Unix Epoch was actually defined on January 1, 1971 and was later revised to January 1, 1970. This epoch allows for a 32-bit unsigned integer to represent approximately 136 years, if each increment since the epoch is equivalent to 1 second. Thus by specifying a 32-bit unsigned integer as the number of seconds since January 1, 1970, you can mark any moment in time until February 7, 2106. If you are limited to signed 32-bit numbers, the maximum date is January 19, 2038.

As stated most system epochs start on January 1, 1970. The question is, how

would you know? The simple answer is to ask the Python time module to tell us using the gmtime() method, gmtime converts the number of seconds (provided as a parameter to the gmtime() method) into the equivalent GMT. To illustrate, let us pass 0 (zero) as the parameter and see what epoch date the method returns.

```
>> > import time
>>>print time.gmtime(0)
time.struct_time(tm_year¼1970, tm_mon¼1, tm_mday¼1, tm_hour¼0,
tm_min¼0, tm_sec¼0, tm_wday¼3, tm_yday¼1, tm_isdst¼0)
```

Similarly, if I wish to know the maximum time value (based upon a maximum limit of an unsigned 32-bit number), I would execute the following code.

```
>> >import time
>>>time.gmtime(0xffffffff)
time.struct_time(tm_year¼2106, tm_mon¼2, tm_mday¼7, tm_hour¼6,
tm_min¼28, tm_sec¼15, tm_wday¼6, tm_yday¼38, tm_isdst¼0)
```

Additionally, you might be wondering, how many seconds have passed since the epoch right now? To determine this, I use the time() method provided by the time module. This method calculates the number of seconds since the epoch.

```
>>>time.time() 1381601236.189
```

As you can see, when executing this command under the Windows version of the time() method, the time() method returns a floating point number not a simple integer, providing subsecond precision.

```
>> >secondsSinceEpoch¼time.time() >>>secondsSinceEpoch
1381601264.237
```

I can then use this time (converted to an integer first) as input to the gmtime() method to determine the date and time in Greenwich.

```
>> >time.gmtime(int(secondsSinceEpoch))
time.struct_time(tm_year¼2013, tm_mon¼10, tm_mday¼12, tm_hour¼18,
tm_min¼7, tm_sec¼44, tm_wday¼5, tm_yday¼285, tm_isdst¼0)
```

The result is October 12, 2013 at 18 hours, 7 minutes, 44 seconds; the weekday is 5 (assuming that the week begins with 0¼Monday, this value would indicate

Saturday); this is the 285th day of 2013; and we are not currently observing daylight savings time. It should be further noted for complete clarity that this time represents UTC/GMT time not local time. Also, it is important to note that the time module is using the computer's system time and timezone settings to calculate the number of seconds since the epoch, not some magical time god. Therefore, if I wish to change what the now is I could do that by simply changing my system clock and then I rerun the previous script.

>> >secondsSinceEpoch¼time.time()
>>>secondsSinceEpoch
1381580629.793
>>>time.gmtime(int(secondsSinceEpoch))
time.struct_time(tm_year¼2013, tm_mon¼10, tm_mday¼12, tm_hour¼12, tm_min¼23, tm_sec¼49, tm_wday¼5, tm_yday¼285, tm_isdst¼0)

This results in our first successful forgery of time.
Now, if I compare GMT/UTC with local time I utilize both the gmtime() and localtime() methods.

>> >import time >>>now¼time.time() >>>now 1381670992.539
>>>time.gmtime(int(now))

time.struct_time(tm_year¼2013, tm_mon¼10, tm_mday¼13, tm_hour¼13, tm_min¼29, tm_sec¼52, tm_wday¼6, tm_yday¼286, tm_isdst¼0)
>>>time.localtime(int(now)) time.struct_time(tm_year¼2013, tm_mon¼10, tm_mday¼13, tm_hour¼9, tm_min¼29, tm_sec¼52, tm_wday¼6, tm_yday¼286, tm_isdst¼1) As you can see this yields:
Local Time: Sunday October 13, 2013 09:29:52
GMT Time: Sunday October 13, 2013 13:29:52

Both the localtime() and gmtime() take into consideration many factors to determine the date and time, one important consideration is the timezone that my system is currently configured to:

I can use Python to provide me with the current timezone setting of my system, by using the following script. Note time.timezone is an attribute rather than a method so it can be read directly.

>> >import time >>>time.timezone 18000

Ok, what does 18,000 mean? We tend to think of timezones being a specific area or zone, which they are. We also tend to think each timezone is exactly 1 hour apart. This is not always true.

For example, some locations around the globe utilize 30-minute offsets from UTC/GMT.

Afghanistan
Australia (both Northern and Southern Territories)
India
Iran
Burma
Sri Lanka

Other locations use 15-minute offsets values related to UTC/GMT. Examples include:

Nepal
Chatham Island in New Zealand
Parts of Western Australia
The 18,000 represents the number of seconds west of UTC and points that are east of UTC are recorded in negative seconds. Since 18,000 represents the number of seconds west of UTC, I can easily calculate the number of hours west of UTC by dividing 18,000 seconds by 60 (seconds in a minute) and then again by 60 (minutes in an hour):

18,000 =60=60¼5hours
This means that my local time is 5 hours west of UTC/GMT. Going back to our comparison then, there should be 5 hours difference between localtime and GMT correct?

Local Time: Sunday October 13, 2013 09:29:52
GMT Time: Sunday October 13, 2013 13:29:52

Not quite! The difference between localtime() and gmtime() in this example only depicts 4 hours difference. It turns out that Sunday October 13, 2013 falls within daylight savings time here in the Eastern U.S. timezone as noted by tm_isdst¼1 found in the local time example, therefore the time difference on this particular date is 4 hours from GMT/UTC correctly reported by the two methods.

Next, you might be wondering how one determines the name of the current time zone set for my system. The time module provides an additional attribute specifically time.tzname that provides a tuple. The first value is the standard time zone designation while the second value is local daylight savings time zone designation.

Here is the example using my local time.

```
>> >import time
>>>time.tzname
('Eastern Standard Time','Eastern Daylight Time')
```

Note, this information is based upon the local operating system internal representation. For example, on a Mac, this returns (“EST,”, “EDT”). Therefore, if we wish to print the current time zone designation of a local system, the following code will provide that for you.

```
import time
# Get the current local time
now¼time.localtime()

# if we are observing daylight savings time # tm_isdst is an attribute that can be examined, # if the value is 0 the current local time is in # standard time observations and if tm_isdst is 1 # then daylight savings time is being observed

if now.tm_isdst:
# print the daylight savings time string
print time.tzname[1]
```



```
else:
# Otherwise print the standard time string print time.tzname[0]
```

Eastern Daylight Time

Another very useful method within the time module is the strftime(). This method provides great flexibility in generating custom strings from the base time structure provided. This allows us to format output without having to extract individual time attributes manually.

For example:

```
import time
print time.strftime("%a, %d %b %Y %H:%M:%S %p",time.localtime())
time.sleep(5)
print time.strftime("%a, %d %b %Y %H:%M:%S %p",time.localtime())
```

This short script prints out the local time value using the strftime method and uses the sleep()method to delay (in this example 5 seconds) and then prints another time string. The result produces the following output:

```
Sun, 13 Oct 2013 10:38:44 AM Sun, 13 Oct 2013 10:38:49 AM
```

The strftime method has great flexibility in the available options, which are shown in Table 6.1.
As you can see the time module has a plethora of methods and attributes and is straightforward to use. For more information on the time module, you can check out the Python Standard Library [PYLIB].

THE NETWORK TIME PROTOCOL

Today, the most widely accepted practice for synchronizing time is to employ the Network Time Protocol (NTP). NTP utilizes the User Datagram Protocol (UDP) to communicate minimalistic timing packets communicated between a server (containing a highly accurate source of time) and a client wishing to synchronize with that time source. (The default server port for the NTP protocol is 123.) The NTP model includes not one, but many easily accessible time servers synchronized to national laboratories. The point of NTP is to convey or distribute time from these servers to clients via the Internet. The NTP protocol standard is governed by the IETF and the Proposed Standard is RFC 5905, titled “Network Time Protocol Version 4: Protocol and Algorithms Specification” [NTP RFC]. Many programs, operating systems, and applications have been developed to utilize this protocol to synchronize time. As you have probably guessed we are going to develop a simple time synchronization Python program and use this to synchronize forensic operations. Instead of implementing the protocol from scratch I am going to leverage a third-party Python Library ntplibto handle the heavy lifting and then compare the results to my local system clock.

Table 6.1 strftime Output Specification

Parameter Definition
%a Abbreviation of weekday string
%A Complete weekday string
%b Abbreviation of month string
%B Complete month name
%c Appropriate date and time representation based on locale %d Day of the month as a decimal number 01-31
%H Hour (24-hour clock) as a decimal number 00-24
%I Hour (12-hour clock) as a decimal number 00-12
%j Day of the year as a decimal number 001-356
%m Month as a decimal number 01-12
%M Minute as a decimal number 00-59
%p AM or PM representation based on locale
%S Second as a decimal number 00-59
%U Week number of the year 00-53 (assumes week begins on Sunday) %w Weekday 0-6 Sunday is 0
%W Week number of the year 00-53 (assumes week begins on Monday)

%x Date representation based on locale
%X Time representation based on locale
%y Year without century as a decimal number 00-99
%Y Year with century as a decimal number
%Z Time zone name

OBTAINING AND INSTALLING THE NTP LIBRARY ntplib

The ntplib is available for download at https://pypi.python.org/pypi/ntplib/ as shown in Figure 6.3 and as of this writing it is currently version 0.3.1. The library provides a simple interface to NTP servers along with methods that can translate NTP protocol fields to text allowing easy access to other key values such as leap seconds and special indicators.

FIGURE 6.3

Python ntplib download page.

Installing the current library is a manual process, but still straightforward and because ntplib is written completely in native Python the library is compatible across platforms (Windows, Linux, and Mac OS X).

The first installation step is to download the installation package in this case a tar. gz file as shown in Figure 6.4.
Once download you must decompress the tar.gz into a local directory. I have unzipped the archive into c:\Python27\Lib\ntplib-0.3.1 (Figure 6.5). Since I am utilizing Python 2.75, decompressing the download in this directory organizes the library with the other installed libs for easy access and update in the future. The installation process for manually installed libraries like ntplib is accomplished by executing the included setup program setup.py that is included in the directory. To perform the installation you simply open a command window as shown in Figure 6.6 and enter the command:

```
Python setup.py install
```

The setup.py program performs all the necessary installation steps. Next, I always like to verify the installation of the new library by opening a Python Shell and importing the library and then by simply typing the name of the library. Python provides basic information regarding the library confirming it is available for use (see Figure 6.7).

As with any module you can always enter the dir(objName) built-in function to obtain details about the properties, classes, and methods that are available for the

FIGURE 6.4
Download of ntplib-0.3.1.tar.gz.

FIGURE 6.5
Decompressed ntplib-0.3.1.

```
C:\Windows\system32\cmd.exe

C:\Python27\Lib>cd ntplib-0.3.1

C:\Python27\Lib\ntplib-0.3.1>dir
 Volume in drive C is HP
 Volume Serial Number is 5EC5-6025

 Directory of C:\Python27\Lib\ntplib-0.3.1

10/16/2013  08:50 PM    <DIR>          .
10/16/2013  08:50 PM    <DIR>          ..
07/03/2013  01:21 PM             1,399 CHANGELOG
10/18/2009  06:55 AM            18,326 COPYING
01/16/2010  06:25 AM             7,802 COPYING.LESSER
09/01/2012  09:49 AM                93 MANIFEST
07/03/2013  01:20 PM            12,818 ntplib.py
07/03/2013  02:38 PM             1,835 PKG-INFO
07/03/2013  01:21 PM             1,670 setup.py
09/01/2012  09:20 AM             3,686 test_ntplib.py
               8 File(s)         47,629 bytes
               2 Dir(s)  168,967,090,176 bytes free

C:\Python27\Lib\ntplib-0.3.1>python setup.py install
running install
running build
running build_py
creating build
creating build\lib
copying ntplib.py -> build\lib
running install_lib
copying build\lib\ntplib.py -> C:\Python27\Lib\site-packages
byte-compiling C:\Python27\Lib\site-packages\ntplib.py to ntplib.pyc
running install_egg_info
Writing C:\Python27\Lib\site-packages\ntplib-0.3.1-py2.7.egg-info

C:\Python27\Lib\ntplib-0.3.1>
```

FIGURE 6.6
Install ntplib.

```
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC
v.1500 32 bit (Intel)]
Type "help", "copyright", "credits" or "license" for
more information.
>>> import ntplib
>>> ntplib
<module 'ntplib' from
'C:\Python27\lib\site-packages\ntplib.pyc'>
>>>
```

FIGURE 6.7 Verifying the installation.

```
>>> dir(ntplib)
['NTP', 'NTPClient', 'NTPException', 'NTPPacket',
'NTPStats', '__builtins__', '__doc__', '__file__',
'__name__', '__package__', '_to_frac', '_to_int',
'_to_time', 'datetime', 'leap_to_text', 'mode_to_text',
'ntp_to_system_time', 'ref_id_to_text', 'socket',
'stratum_to_text', 'struct', 'system_to_ntp_time', 'time']
>>>
>>>
>>>
>>>
```

FIGURE 6.8 dir(ntplib) results.

object (see Figure 6.8). For even more information, you can use the help(objName) built-in function.

WORLD NTP SERVERS

Examining the methods and properties may seem a bit confusing at first; however, using the library module is in fact quite simple. Simply put we want to create an ntp client capable of accessing a specified ntp server to obtain a third-party source of time. In the United States, NIST manages a list of time servers that can be accessed to obtain “root” time (Figure 6.9).

Updated lists can be found at: http://tf.nist.gov/tf-cgi/servers.cgi. In Europe, you can find an active list of NTP servers at the NTP Pool Project. Figure 6.10 shows a screenshot from their home page at http://www.pool.ntp.org/ zone/europe.

Name	IP Address	Location	Status
nist1-ny.ustiming.org	64.90.182.55	New York City, NY	
nist1-nj.ustiming.org	96.47.67.105	Bridgewater, NJ	all services available
nist1-nj2.ustiming.org	165.193.126.229	Weehawken, NJ	all services available
nist1-ny2.ustiming.org	216.171.112.36	New York City, NY	All services available
nist1-pa.ustiming.org	206.246.122.250	Hatfield, PA	All services available
time-a.nist.gov	129.6.15.28	NIST, Gaithersburg, Maryland	All services busy, not recommended
time-b.nist.gov	129.6.15.29	NIST, Gaithersburg, Maryland	All services busy, not recommended
time-c.nist.gov	129.6.15.30	NIST, Gaithersburg, Maryland	All services available
time-d.nist.gov	2610:20:6F15:15::27	NIST, Gaithersburg, Maryland	All services via IPV6
nist1.aol-va.symmetricom.com	64.236.96.53	Reston, Virginia	All services available
nist1-macon.macon.gatech.edu	98.175.203.200	Macon, Georgia	All services available
nist1-atl.ustiming.org	64.250.177.145	Atlanta, Georgia	All services available
wolfsittime.com	207.223.123.18	Birmingham, Alabama	All services available
nist1-chi.ustiming.org	216.171.120.36	Chicago, Illinois	All services available

FIGURE 6.9
Partial list of NIST time servers.

FIGURE 6.10
European NTP Pool Project.

For those of you wishing to obtain your time from the U.S. Naval Observatory or

USNO, you might be interested to know that for many years, the USNO has provided access to NTP servers' aptly named tick and tock. For more information on the U.S. Naval Observatory you can visit: http://www.usno.navy.mil/USNO.

NTP Client Setup Script 179
NTP CLIENT SETUP SCRIPT
To setup a Python ntp client using the ntplib is a simple process: import ntplib # import the ntplib
import time # import the Python time module
# url of the closest NIST certified NTP server
NIST¼'nist1-macon.macon.ga.us'
# Create NTP client object using the ntplib
ntp¼ntplib.NTPClient()
# initiate an NTP client request for time
ntpResponse¼ntp.request(NIST)
# Check that we received a response

if ntpResponse:
# obtain the seconds since the epoch from response
nistUTC¼ntpResponse.tx_time
print'NIST reported seconds since the Epoch :', print nistUTC

else:
print'NTP Request Failed'

Program Output
NIST reported seconds since the Epoch : 1382132161.96
Now that we can obtain the seconds since the epoch, we can use the Python Standard Library time module to display the time in either local or GMT/UTC time, and can even compare the current NTP time to our own local system clock as shown below.

CODE LISTING 6.1
import ntplib import time
NIST¼'nist1-macon.macon.ga.us'
ntp¼ntplib.NTPClient()
ntpResponse¼ntp.request(NIST)

if (ntpResponse):
now¼time.time()
diff¼now-ntpResponse.tx_time print'Difference :',

Continued print diff,
print'seconds'

print'Network Delay:', print ntpResponse.delay
print'UTC: NIST :'þtime.strftime("%a, %d %b %Y %H:%M:%S +0000", time.gmtime(int(ntpResponse.tx_time)))
print 'UTC: SYSTEM : 'þtime.strftime("%a, %d %b %Y %H:%M:%S +0000", time.gmtime(int(now)))
else:
print'No Response from Time Service'

Program Output
Difference : 3.09969758987 seconds Network Delay : 0.0309739112854
UTC NIST : Fri, 18 Oct 2013 21:48:48 +0000 UTC SYSTEM : Fri, 18 Oct 2013 21:48:51 +0000

Notable observations:
(1) It is important that you obtain the local system time immediately after obtaining the time from the time server (in my case from NIST):
ntpResponse¼ntp.request(NIST) if (ntpResponse):
now¼time.time()
This ensures that our comparison with the local time maintained by our system clock is influenced by the smallest amount of processing.

(2) It is also important to consider the network delay reported by the NTP client. In this example, the delay was a little over 30 milliseconds (specifically 0.0309739112854 seconds), however, in certain situations the delay can be much longer based on the time of day, your network connection speed, and other network latency issues.

(3) In this example, I calculated a difference of 3.09969758987 seconds. This means that my local system clock is running 3þseconds faster than that of the NIST server.
diff¼now-ntpResponse.tx_time If the resulting value was negative instead of positive, I can conclude that my system clock is running behind that reported by

NIST.
(4) Finally, the ntplib client operations are synchronous. In other words, once I make the request:
ntpResponse¼ntp.request(NIST) the code does not continue processing until the request is fulfilled or fails just like any method or function call.
Additional Resources 181
CHAPTER REVIEW

In this chapter, I explained the basics of time and computing. This includes a better understanding of the epoch, the origins of modern time keeping along with a bit of history on time. I then took a deep dive into the Standard Library module time and I explained and demonstrated many methods and properties associated with the module. I developed several short Python scripts to give you a feel for how to apply and interpret the results of time processing.

I then provided an overview of the NTP that provides us with the ability to synchronize time with National Measurement sources. Next, I walked you through the installation and setup of a Python NTP Library ntplib. I then experimented with the module to setup a NTP client that interfaces with a network-based time source. I wrote a script that would compare and calculate the difference between my local system clock and that of a NIST time server.

SUMMARY QUESTIONS

1. Back in Chapter 3, I developed a program that walked the files system, hashed the files, and recorded the modified, access, and created (MAC) times of each file. Modify that program using the time module to convert the MAC times from local time to GMT/UTC time values.

2. Expanding on Code Listing 6.1, choose five additional time servers throughout the world and devise a method to compare and contrast the time reported by each in relationship to your local system clock. Also, record the network delay values from each. Note, you should execute multiple runs and average the results.

3. What other time sources should be carefully examined and/or normalized during a forensic investigation?
Additional Resources
Proving the integrity of digital evidence with time. Int J Digital Evid 2002;1(1)

1–7.

The Python Standard Library. http://docs.python.org/2/library/time.html?highlight¼ time#time.
The Network Time Protocol Version 4: Protocol and Algorithms Specification. http://tools. ietf.org/html/rfc5905.
Longitude. Longitude: the true story of a Lone Genius who solved the greatest scientific problem of his time. New York, NY: Walker & Co.; 1995.

CHAPTER

Using Natural Language Tools in Forensics 7

CHAPTER CONTENTS

What is Natural Language Processing? 183
Dialog-Based Systems 184
Corpus 184
Installing the Natural Language Toolkit and Associated Libraries 185
Working with a Corpus 185
Experimenting with NLTK 186
Creating a Corpus from the Internet 193
NLTKQuery Application 194
NLTKQuery.py 195
_classNLTKQuery.py 196
_NLTKQuery.py 198
NLTKQuery Example Execution 199
NLTK Execution Trace 199
Chapter Review 202
Summary Questions 202
Additional Resources .....

WHAT IS NATURAL LANGUAGE PROCESSING?

Before I begin, Natural Language needs to be defined. Back in 1987, I read the book Introduction to Natural Language Processing as part of a research project I was working on with Dr. James Cook at the Illinois Institute of Technology Research Institute (IITRI). My wife Janet was also working on Natural Language projects at IITRI with Dr. Larry Jones. Her work employed Natural Language as part of a decision engine that could diagnose and better understand the behaviors of cattle. The book we referenced had the best definitions that I have seen for Natural Language, therefore I will share that excerpt with you here.

"Natural Language is any language that humans learn from their environment and use to communicate with each other. Whatever the form of the communication, Natural Languages are used to express our knowledge and emotions and to convey our responses to other people and to our surroundings." [Harris]

Python Forensics 183© 2014 Elsevier Inc. All rights reserved.

Obviously, like all good definitions they stand the test of time. Since the Internet really did not materialize into a recognizable form until the late 1980s, the concept of expressing emotion and knowledge digitally did not include Facebook, Twitter, Emoticons, Blogging, Texting, Sexting, BitStrips, or even Internet-based e-mail! However, the definition certainly provides coverage for these latest forms of Natural Language expression. Extracting meaning (semantics) from Natural Language is quite a bit more difficult than it looks.

Judging the effectiveness of Natural Language systems that attempt to interpret, understand, and respond accordingly falls to the Turing Test, named after Alan Turing based on his 1950 paper "Computing Machinery and Intelligence." In his paper, Turing asks the proactive question, "Can machines think?" [Turing]

Dialog-based systems

The goal of dialog-based systems is to respond to a user input naturally, meaning that we cannot distinguish between human or machine-based responses. Some may be familiar with systems that are moving toward this capability. For example, here is a dialog between me and the Sync system in my car:

Me: I'm hungry.
Sync: Ok, Where would you like to search nearby, along your route or in a city?
Me: Nearby.
Sync: How would you like to search—by name or by category? Me: By category.
Sync: Ok, what category?
Me: Italian.
Sync: Ok, please select from the list on your screen, for example say Line 1. Me: Line 4.
Sync: Ok, Rossi's Restaurant. Would you like to make a call or set as destination? Me: Make a call.
Sync: Ok: Calling Rossi's Restaurant.....

Although useful, it certainly does not meet the Turing standard because I know that I am talking to a computer not a human. However, Sync does translate the spoken word into text and then responds in a timely fashion and as a result the system usually gets me where I want to go or provides me the information I am seeking.

Corpus

One of the key elements needed to perform Natural Language Processing (NLP) is a corpus. A corpus is a large volume of linguistic samples that is relevant to a specific genre. For example, the Guttenberg Project [Gutenberg] that is available at http:// www.gutenberg.org/ contains over 40,000 books in electronic form.

One of the challenges in creating meaningful examples in this chapter was establishing a corpus. Then I needed to demonstrate the use of the corpus to create these

examples that will serve as a catalyst for the development of new applications of NLP within the field of forensic science. Bear in mind, depending on the application you intend for NLP, more than one genre may be necessary.

INSTALLING THE NATURAL LANGUAGE TOOLKIT AND ASSOCIATED LIBRARIES

As with other third-party libraries and modules, installation is available for Windows, Linux, and Mac. The Natural Language Toolkit (NLTK) library is free

and can be easily obtained online, from nltk.org. Installation of NLTK does require the installation of other dependency libraries including Numpy and PyYAML. Figure 7.1 depicts the nltk.org installation page with easy to follow instructions. Once you have installed everything, you can verify the installation through your favorite Python Shell by typing:

Import NLTK

WORKING WITH A CORPUS

The first step in working with a corpus is to either load an existing corpus that is included with NLTK or create your own from local files or the Internet. Once this is done, there are a variety of powerful operations you can perform on the corpus,

FIGURE 7.1
nltk.org Installation url.

Table 7.1 Introductory List of NLTK Operations

Method Description raw()
word_tokenize()
collocations()

concordance()
findall(search)
index(word)
similar(word)

vocab() Extracts the raw contents of the corpus. The method returns the type: Python “str”
Removes white space and punctuation and then creates a list of the individual tokens (basically words). The method returns the type: Python “list”
Words often appear consecutively within a text and these provide useful information to investigators. In addition, once detected they can be used to find associations between word occurrences. This is accomplished by calculating the frequencies of words and their connections with other words. The collocations method performs all of this in a single method
This method provides the ability to generate every occurrence of a specific word along with the context (how the word was used in a specific sentence, for example)
This method can be used for simple and even regular expression searches of the corpus text
This method provides an index to the first occurrence of the word provided
Note this is not synonyms, rather the method provides distribution similarities, stated simply it identifies other words which appear in the same “context” as the specified word
Produces a comprehensive vocabulary list of the text submitted

operations that would be both complicated and time-consuming without the aid of a toolset like NLTK. Table 7.1 depicts some of the key operations that can be performed on a corpus, I will be demonstrating these in code later in this chapter.

EXPERIMENTING WITH NLTK

In this section, I am going to invoke the methods in Table 7.1 and create a simple program to first create a new text corpus, and then examine that corpus with these methods. First, a little background on the corpus I am going to create.

Jack Walraven maintains the web site http://simpson.walraven.org/. At that site, Jack has organized the transcripts of the O.J. Simpson trial. For this experiment, I downloaded the Trial Transcripts for January 1995 (you can of course download as many months and excerpts as you like). There are nine Trial

Transcript text files stored there for the January 1995 proceedings, and they represent transcripts for the dates of January 11-13, 23-26, 30, and 31. I created a directory on my Windows system, c:\simpson\ , to hold each of the separate files. My goal is to use these files to create a new text corpus that includes all of these files. The best way to illustrate how this works is to walk you through some sample code that deals with each aspect. I have annotated the code to make sure you understand each step of the process.

# NLTK Experimentation This first section imports the modules necessary for the experiment. The most important is the newly installed NLTK module
from __future__ import division
import nltk

Next, I'm going to import the PlaintextCorpusReader from the NLTK module. This method will allow me to read in and ultimately create a text corpus that I can work with.

from nltk.corpus import PlaintextCorpusReader

The first thing I need to specify is the location of the files that will be included in the corpus. I have placed those files in c:\simpson\. Each of the 9 files are stored there and each is a text document.

rootOfCorpus ¼ "c:\\simpson\\"

Now I 'm going to use the PlainTextCorpusReader method to collect all the files in the directory I specified in the variable rootOfCorpus. I specify all the files by the second parameter '.*'. If you have directories that had multiple file types and you only wanted to include the text documents you could have specified '.txt'. The result of this call will be an NLTK Corpus Object named newCorpus.

newCorpus ¼ PlaintextCorpusReader(rootOfCorpus,'.*')
print type(newCorpus)
The print type(newCorpus) produces the following output providing me with the NLTK type of the newCorpus.
<class'nltk.corpus.reader.plaintext.PlaintextCorpusReader'> I can also print the file identification of each file that makes up the new corpus by using the fileids() method.
print newCorpus.fileids()

['Trial-January-11.txt','Trial-January-12.txt','Trial-January-13.txt', 'Trial-January-23.txt','Trial-January-24.txt','Trial-January-25.txt', 'Trial-January-26.txt','Trial-January-30.txt','Trial-January-31.txt']

I can also, determine the absolute paths of the individual documents contained in the new corpus, by using the abspaths() method.
print newCorpus.abspaths()

[FileSystemPathPointer('c:\\simpson\\Trial-January-11.txt'),
FileSystemPathPointer('c:\\simpson\\Trial-January-12.txt'),
FileSystemPathPointer('c:\\simpson\\Trial-January-13.txt'),
FileSystemPathPointer('c:\\simpson\\Trial-January-23.txt'),
FileSystemPathPointer('c:\\simpson\\Trial-January-24.txt'),
FileSystemPathPointer('c:\\simpson\\Trial-January-25.txt'),
FileSystemPathPointer('c:\\simpson\\Trial-January-26.txt'),
FileSystemPathPointer('c:\\simpson\\Trial-January-30.txt'),
FileSystemPathPointer('c:\\simpson\\Trial-January-31.txt')]

Now I can leverage the newCorpus object to extract the raw text which represents the complete collection of files contained in the rootOfCorpus directory specified. I can then determine the length or size of the combination of the 9 trial transcript files.

rawText ¼ newCorpus.raw()
print len(rawText)
This produces the output:
2008024

I can now apply the nltk .Text module methods in order to better understand and interpret the content of the corpus. I start this process by tokenizing the rawText of the corpus. This produces tokens (mainly words, but also includes numbers and recognized special character sequences).

tokens ¼ nltk.word_tokenize(rawText)

Tokenization creates a standard Python list that I can now work with using both standard Python language operations and of course any methods associated with list objects. For example I can use len (tokens) to determine the number of tokens that were extracted from the text.

print len(tokens)

Which produces the result for this corpus of 401 ,032 tokens.
401032

Since this is now a simple list , I can display some or a portion of the contents. In this example I print out the first 100 list elements representing the first 100 tokens returned from the tokenization process.

print tokens[0:100]

```
[
'*LOS','ANGELES',',','CALIFORNIA',';','WEDNESDAY',',','JANUARY','11',',','1
'9:05','A.M.*','DEPARTMENT','NO.','103','HON.','LANCE','A.','ITO',',','JUDGE',
'APPEARANCES',':','(','APPEARANCES','AS','HERETOFORE','NOTED',',','DE
'DISTRICT','ATTORNEY','HANK',';','ALSO','PRESENT','ON','BEHALF','OF','S
',','MS.','PAMELA','W.','WITHEY',',','ATTORNEY-AT-LAW.',')','(','JANET','M.',
'MOXHAM',',','CSR','NO.','4855',',','OFFICIAL','REPORTER.',')','(','CHRISTINE
'M.','OLSON',',','CSR','NO.','2378',',','OFFICIAL','REPORTER.',')','(','THE',
'FOLLOWING','PROCEEDINGS','WERE','HELD','IN','OPEN','COURT',',','OUT
'PRESENCE','OF','THE','JURY',':',')','*THE','COURT',':','*','ALL','RIGHT.','THE'
'SIMPSON']
```

Next , I want to create a text that I can apply the NLTK text methods to. I do this by creating NLTK Text Object textSimpson using the Text method to the tokens extracted by the tokenization process.

textSimpson ¼ nltk.Text(tokens)
print type(textSimpson)

As expected the object type is nltk .text.Text <class'nltk.text.Text'>

I would like to know the size of the vocabulary used across the corpus .I know the number of tokens, but there are of course duplicates. Therefore, I use the Python set method to obtain the unique set of vocabulary present in the text in the trial transcript.

vocabularyUsed ¼ set(textSimpson)
print len(vocabularyUsed)
12604

Which produces 12 ,604 unique tokens from the total of 401,032 a much more manageable number. From this I would like to see all the unique tokens, and I can accomplish that by sorting the set.

```
print sorted(set(textSimpson))

'ABERRANT','ABIDE','ABIDING','ABILITIES','ABILITIES.','ABILITY','ABIL
'ABLAZE','ABLE','ABOUT','ABOUT.','ABOVE','ABRAHAM','ABROGATE','A
'ABRUPT','ABRUPTLY','ABSENCE','ABSENCE.','ABSENT','ABSOLUTE','AB
'ABSOLUTELY.','ABSORB','ABSTRACT','ABSTRACT.','ABSURD.','ABSURD
'ABUDRAHM.','ABUNDANCE','ABUNDANT','ABUNDANTLY','ABUSE','AB
BATTERED','ABUSED','ABUSER','ABUSER.','ABUSES','ABUSES.','ABUSIN
'ABUSIVE.',

...
... skipping about 10,000 tokens
...

'WRITTEN','WRITTEN.','WRONG','WRONG.','WRONGFULLY','WRONGLY',
'YAMAUCHI','YAMAUCHI.','YARD','YARD.','YARDS','YEAGEN','YEAH','YE
'YEAR.','YEARS','YEARS.','YELL','YELLED','YELLING','YELLING.','YELLO
'YES','YES.','YESTERDAY','YESTERDAY.','YET','YET.','YIELD','YORK','YOR
"YOU'LL", "YOU'RE",
"YOU'VE","YOU.','YOUNG','YOUNGER','YOUNGSTERS','YOUR',
'YOURS.','YOURSELF','YOURSELF.','YOURSELVES','YOUTH','YUDOWITZ
'Z','ZACK','ZACK.','ZEIGLER','ZERO','ZLOMSOWITCH',
"ZLOMSOWITCH'S", 'ZLOMSOWITCH.','ZOOM'
```

Next, I can determine how many times a particular word appears in the Trial Transcript. I accomplish using the count() method of the nltk.Text object type.

```
myWord ¼ "KILL"
textSimpson.count(myWord)
84
```

This produces a count of the 84 times that the word KILL appeared in the Trial Transcription for the 9 days included in this corpus,during the month of January 1995. Up to this point all of this is pretty straight-forward. Let’s take a look at a couple of the more advanced methods included in the nltk.Text Module. I’m going to start with the collocations method. This will provide me with a list of

words that occur together statistically often in the text.

```
print textSimpson.collocations()
```

Building collocations list
*THE COURT; *MS. CLARK; *MR. COCHRAN; MR. SIMPSON; NICOLE BROWN; *MR.DARDEN; OPENING STATEMENT; LOS ANGELES; MR. COCHRAN; DETECTIVE FUHRMAN; DISCUSSION HELD; WOULD LIKE; *MR. DOUGLAS; BROWN SIMPSON;THANK YOU.; MR. DARDEN; DEPUTY DISTRICT; FOLLOWING PROCEEDINGS; DISTRICT ATTORNEYS.; MISS CLARK

As you can see , the collocation list generated makes a great deal of sense. These pairings would naturally occur more often that others during the proceedings. Next, I want to generate a concordance from the transcript for specific words that might be of interest. NLTK will produce each occurrence of the word along with context (in other words the sentence surrounding the concordance word). A second optional parameter indicates the size of the window of surrounding words you would like to see. I will demonstrate a couple examples.

I will start with the word “KILL”, this produces 84 matches and I have included the first 6.

```
myWord ¼ "KILL"
print textSimpson.concordance(myWord)
```

Displaying 6 of 84 matches:
WAS IN EMINENT DANGER AND NEEDED TO KILL IN SELF-DEFENSE. BUT THE ARIS COURT

R OCCURRED. "I KNOW HE'S GOING TO KILL ME. I WISH HE WOULD HURRY UP AND GET FLICTED HARM WAY BEYOND NECESSARY TO KILL THE VICTIM. AND THIS IS A QUOTE FROM

'M GOING TO HURT YOU , I'M GOING TO KILL YOU , I'M GOING TO BEAT YOU."THO
HAVE HER AND NO ONE ELSE WILL IS TO KILL HER. THAT IS CLEAR IN THE RESEARCH.
WAS A FIXED PURPOSE , FULL INTENT TO KILL , WHERE THE DEFENDANT LITERALLY WENT

If I use the word “GLOVE”, the code produces 92 matches; I have included the

first 6:
myWord¼ "GLOVE"
print textSimpson.concordance(myWord)
Displaying 6 of 92 matches:
CE DEPARTMENT PLANTED EVIDENCE , THE GLOVE AT ROCKINGHAM. NOW , THAT OF COURSE
HAT A POLICE DETECTIVE WOULD PLANT A GLOVE , AND IT MADE HOT NEWS AND THE DEFEN
R THIS POLICE DETECTIVE PLANTED THIS GLOVE AT ROCKINGHAM. NOW , YOUR HONOR , BE
DETECTIVE FUHRMAN'S RECOVERY OF THE GLOVE AND AS – INSOFAR AS IT RELATES TO T
HEY SAW THE LEFT-HANDED BULKY MAN'S GLOVE THERE AT THE FEET OF RONALD GOLDMAN.
E BUNDY CRIME SCENE. THEY SAW A LONE GLOVE , A SINGLE GLOVE AT THE FEET OF RONA

NLTK also provides more sophisticated methods that can operate on text corpus . For example the ability to identify similarities of word usage. The method identifies words that are used within similar contexts.

myWord¼ "intent"
print textSimpson.similar(myWord)

Building word-context index.. .
time cochran court evidence and house it blood defense motive that jury other people the this witnesses case defendant discovery

if I change the word to “victim” the results are as follows:

Building word-context index .. .
court defendant jury defense prosecution case evidence and people record police time relationship house question statement tape way crime glove

I can also utilize NLTK to produce a comprehensive vocabulary list and frequency distribution that covers all the tokens in the corpus. By using this method across documents, one can derive tendencies of the writer or creator of a document. The vocab method returns an object based on the class nltk.probability.FreqDist as shown here.

```
simpsonVocab ¼ textSimpson.vocab()
type(simpsonVocab)
<class'nltk.probability.FreqDist'>
```

Thus the simpsonVocab object can now be utilized to examine the frequency distribution of any and all tokens within the text. By using just one of the methods, for example, simpsonVocab.items(), I can obtain a sorted usage list (most used first) of each vocabulary item.

```
simpsonVocab.items()
<bound method FreqDist.items of <FreqDist with 12604 samples and 401032
outcomes>>

[('THE', 19386), (',', 18677), ('TO', 11634), ('THAT', 10777), ('AND', 8938),
(':', 8369), ('OF', 8198), ('*', 7850), ('IS', 6322), ('I', 6244), ('A', 5590), ('IN',
5456), ('YOU', 4879), ('WE', 4385), ('THIS', 4264), ('IT', 3815), ('COURT',
3763), ('WAS', 3255), ('HAVE', 2816), ('–', 2797), ('?', 2738), ('HE', 2734),
("'S", 2677), ('NOT', 2417), ('ON', 2414), ('THEY', 2287), ('*THE', 2275),
('BE', 2240), ('ARE', 2207), ('YOUR', 2200), ('WHAT', 2122), ('AT', 2112),
('WITH', 2110),

...
... Skipping to the bottom of the output for brevity
...

('WRENCHING', 1), ('WRESTLING.', 1), ('WRISTS.', 1), ('WRITERS', 1),
('WRONGLY', 1), ('X-RAYS', 1), ('XANAX', 1), ('XEROXED.', 1),
('XEROXING.', 1), ('YAMAUCHI.', 1), ('YARD.', 1), ('YARDS', 1),
('YEAGEN', 1), ('YELL', 1), ('YELLING.', 1), ('YEP.', 1), ('YIELD', 1),
('YOUNGSTERS', 1), ('YOURS.', 1), ('YUDOWITZ', 1), ('YUDOWITZ.', 1),
('Z', 1), ('ZEIGLER', 1), ('ZERO', 1), ("ZLOMSOWITCH'S", 1)]
```

Note: A nice by-product of the vocab method is when different vocabulary items have the same number of occurrences, they are sorted alphabetically.

CREATING A CORPUS FROM THE INTERNET

In many cases, the text that we wish to examine is found on the Internet. By combining Python, the Python Standard Library urllib, and NLTK we can

perform the same types of analysis of online documents. I will start with a simple text document downloaded from the Guttenberg Project to illustrate.

I start by importing the necessary modules in this case NLTK and the urlopen module from the urllib. Note: As you can see, I can selectively bring in specific modules from a library instead of loading the entire library.

```
>>> import nltk
>>> from urllib import urlopen
```

Next, I specify the URL I wish to access and call the urlopen.read method.

```
>>> url ¼ "http://www.gutenberg.org/files/2760/2760.txt"
>>> raw ¼ urlopen(url).read()
```

As expected this returns the type ‘str’.

Note : since I’m entering these commands from the Python Shell and interrogating the output directly, I’m not using the same try/except care that I would use if this example was actual program execution. You can of course add that if you’d like.

```
>>> type(raw)
<type'str'>
```

Now that I have confirmed that the result is a string, I check the length that was returned. In this case 3.6MB

```
>>> len(raw)
3608901
```

To get a feel for what is contained in this online source , I simply use the standard Python string capabilities to print out the first 200 characters, this produces the following output.

```
>>> raw[:200]
```

’CELEBRATED CRIMES, COMPLETE\r\n\r\n\r\nThis eBook is for the use of anyone anywhere at no cost and with almost\r\nno restrictions whatsoever. You may copy it, give it away or re-use it\r\n’

Now I can perform any of the NLTK module capabilities, I have included a couple here to give you a feel for the text that I downloaded.

```
>>> tokenList ¼ nltk.word_tokenize(raw)
```

```
>>> type(tokenList)
<type'list'>
>>> len(tokenList)
707397
>>> tokenList[:40]

[' CELEBRATED', 'CRIMES', ',', 'COMPLETE', 'This', 'eBook', 'is', 'for', 'the',
'use', 'of', 'anyone', 'anywhere', 'at', 'no', 'cost', 'and', 'with', 'almost', 'no',
'restrictions', 'whatsoever.', 'You', 'may', 'copy', 'it', ',', 'give', 'it', 'away', 'or', 're-
use', 'it', 'under', 'the', 'terms', 'of', 'the', 'Project', 'Gutenberg']
```

Many other NLTK methods, objects, and functions exist to allow the exploration of Natural Language. For now, you have learned the basics using the test-then code method. Now I will create a simple application that will make it much easier to both access NLTK capabilities using these same methods and provide a baseline for you to extend the capability.

NLTKQuery APPLICATION

In order to make the interface with NLTK both easier and extensible, I have built the NLTKQuery application. The application has three source files:

Source File Description

NLTKQuery.py

_classNLTKQuery.py

_NLTKQuery.py Serves as the main program loop for interfacing with NLTK

This module defines a new class that when instantiated and used properly will allow access to NLTK methods in a controlled manner

This module provides support functions for the NLTKQuery main program loop, mainly to handle user input and menu display

The program provides a simplified user interface access to NLTK methods without forcing the user to understand how the NLTK library works. All the user needs to do is setup a directory that contains the file or files that they wish to include in a corpus. The application will create the corpus based on the directory path provided and then allow the user to interact, or better stated, query the corpus.

The contents of the three source files are listed here:

NLTKQuery.py

```
#
# NLTK QUERY FRONT END
# Python-Forensics
# No HASP required
#

import sys
import _NLTKQuery
print "Welcome to the NLTK Query Experimentation"
print "Please wait loading NLTK ... "

import _classNLTKQuery
oNLTK ¼ _classNLTKQuery.classNLTKQuery()

print
print "Input full path name where intended corpus file or files are stored" print
"Note: you must enter a quoted string e.g. c:\\simpson\\ " print
userSpecifiedPath ¼ raw_input("Path: ")

# Attempt to create a text Corpus
result¼ oNLTK.textCorpusInit(userSpecifiedPath)
if result¼¼ "Success": menuSelection ¼ -1
while menuSelection !¼ 0:

if menuSelection !¼ -1:
print
s ¼ raw_input('Press Enter to continue.. .')

menuSelection ¼ _NLTKQuery.getUserSelection() if menuSelection ¼¼ 1:
oNLTK.printCorpusLength() elif menuSelection ¼¼ 2:
oNLTK.printTokensFound() elif menuSelection ¼¼ 3: oNLTK.printVocabSize()
elif menuSelection ¼¼ 4: oNLTK.printSortedVocab() elif menuSelection ¼¼ 5:
oNLTK.printCollocation()

elif menuSelection ¼¼ 6:
oNLTK.searchWordOccurence()
elif menuSelection ¼¼ 7:
oNLTK.generateConcordance()
```

elif menuSelection ¼¼ 8:
oNLTK.generateSimiliarities()
elif menuSelection ¼¼ 9: oNLTK.printWordIndex() elif menuSelection ¼¼ 10:
oNLTK.printVocabulary()

elif menuSelection ¼¼ 0: print "Goodbye"
print

elif menuSelection ¼¼ -1: continue

else:
print "unexpected error condition" menuSelection ¼ 0

else: print "Closing NLTK Query Experimentation"
_classNLTKQuery.py

#
NLTK QUERY CLASS MODULE # Python-Forensics
No HASP required

import os
import sys
import logging
import nltk
#Standard Library OS functions

# Standard Library Logging functions # Import the Natural Language Toolkit
from nltk.corpus import PlaintextCorpusReader #Import the PlainText

CorpusReader Module # NLTKQuery Class class classNLTKQuery:

def textCorpusInit(self, thePath): # Validate the path is a directory
if not os.path.isdir(thePath):

return "Path is not a Directory" # Validate the path is readable
if not os.access(thePath, os.R_OK):

return "Directory is not Readable" # Attempt to Create a corpus with all .txt files
found in the directory
try:

```
self.Corpus¼ PlaintextCorpus Reader(thePath,'.*')
print "Processing Files : " print self.Corpus.fileids() print "Please wait ¨ ."
self.rawText ¼ self.Corpus.raw() self.tokens ¼ nltk.word_tokenize (self.rawText)
self.TextCorpus¼nltk.Text(self. tokens)

except:
return "Corpus Creation Failed" self.ActiveTextCorpus ¼ True return "Success"
def printCorpusLength(self):
print "Corpus Text Length: ", print len(self.rawText)

def printTokensFound(self):
print "Tokens Found: ",
print len(self.tokens)

def printVocabSize(self):
print "Calculating …"
print "Vocabulary Size: ",
vocabularyUsed ¼ set(self.TextCorpus) vocabularySize ¼ len(vocabularyUsed)
print vocabularySize

def printSortedVocab(self):
print "Compiling ¨ ."
print "Sorted Vocabulary ", print sorted(set(self.TextCorpus))

def printCollocation(self):
print "Compiling Collocations ..." self.TextCorpus.collocations()

def searchWordOccurence(self):
myWord ¼ raw_input("Enter Search Word : ") if myWord:

wordCount ¼ self.TextCorpus.count (myWord)
print myWord+" occured: ", print wordCount,
print " times"

else:
print "Word Entry is Invalid" def generateConcordance(self):

myWord¼ raw_input("Enter word to Concord : ") if myWord:
```

```
self.TextCorpus.concordance
(myWord)
else:
print "Word Entry is Invalid"

def generateSimiliarities(self): myWord¼ raw_input("Enter seed word : ") if
myWord:
self.TextCorpus.similar(myWord) else:
print "Word Entry is Invalid" def printWordIndex(self):

myWord ¼ raw_input("Find first occurrence of what Word? : ")
if myWord:

wordIndex ¼ self.TextCorpus.index (myWord)
print "First Occurrence of: " + myWord + "is at offset: ",
print wordIndex

else:
print "Word Entry is Invalid"

def printVocabulary(self):
print "Compiling Vocabulary Frequencies", vocabFreqList ¼
self.TextCorpus.vocab() print vocabFreqList.items()
```

_NLTKQuery.py

```
#
# NLTK Query Support Methods
# Python-Forensics
# No HASP required
#

# Function to print the NLTK Query Option Menu def printMenu():
print "¼¼¼¼¼¼¼¼¼¼¼NLTK Query Options ¼¼¼¼¼¼¼¼¼¼" print "[1] Print
Length of Corpus"
print "[2] Print Number of Token Found"
print "[3] Print Vocabulary Size"
print "[4] Print Sorted Vocabulary"
print "[5] Print Collocation"
print "[6] Search for Word Occurrence"
```

```
print "[7] Generate Concordance"
print "[8] Generate Similarities"
print "[9] Print Word Index"
print "[10] Print Vocabulary"
print
print "[0] Exit NLTK Experimentation"
print

# Function to obtain user input
def getUserSelection ():
printMenu ()

try:
menuSelection ¼ int(input('Enter Selection (0-10) >> ') )
except ValueError:
print'Invalid input. Enter a value between 0 -10 .' return -1

if not menuSelection in range(0, 11):
print'Invalid input. Enter a value between 0 - 10.' return -1

return menuSelection
```

NLTKQuery example execution

Executing NLTKQuery from the command line for Windows, Linux, or MAC is the same. Simply use the command Python NLTKQuery.py and then follow the onscreen instructions and menu prompts.

NLTK execution trace

Note for brevity some of the output was edited where noted. I also left out the re-display of the menu options.

```
C:\Users\app\ Python NLTKQuery.py
Welcome to the NLTK Query Experimentation
Please wait loading NLTK ...
Input full path name where the intended corpus file or files are stored Format for
Windows e.g. c:\simpson\
Path: c:\simpson\
Processing Files:
['Trial-January-11.txt', 'Trial-January-12.txt', 'Trial-January-13.txt', 'Trial-
January-23.txt', 'Trial-January-24.txt', 'Trial-January-25.txt', 'Trial-January-
```

26.txt', 'Trial
-January-30.txt', 'Trial-January-31.txt']

¼¼¼¼¼¼¼¼¼¼NLTK Query Options ¼¼¼¼¼ [1] Print Length of Corpus
[2] Print Number of Token Found [3] Print Vocabulary Size
[4] Print Sorted Vocabulary
[5] Print Collocation
[6] Search for Word Occurrence [7] Generate Concordance
[8] Generate Similarities
[9] Print Word Index
[10] Print Vocabulary

[0] Exit NLTK Experimentation
Enter Selection (0-10) >> 1 Corpus Text Length: 2008024 Enter Selection (0-10) >> 2 Tokens Found: 401032

Enter Selection (0-10) >> 3 Calculating.. .
Vocabulary Size: 12604

Enter Selection (0-10) >> 4
Compiling .. . Sorted Vocabulary
'ABYSMALLY', 'ACADEMY', 'ACCENT', 'ACCEPT', 'ACCEPTABLE',
'ACCEPTED', 'ACCEPTING', 'ACCESS', 'ACCESSIBLE', 'ACCIDENT',
'ACCIDENT.', 'ACCIDENTAL', 'ACCIDENTALLY', 'ACCOMMODATE',
.
. Edited for brevity
.
'YOUNGER', 'YOUNGSTERS','YOUR','YOURS.',
'YOURSELF','YOURSELF.','YOURSELVES', 'YOUTH', 'YUDOWITZ',
'YUDOWITZ.', 'Z', 'ZACK', 'ZACK.',
"ZLOMSOWITCH'S","ZLOMSOWITCH.', 'ZOOM', ""]

Enter Selection (0-10) >> 5
Compiling Collocations...
Building collocations list
*THE COURT;*MS. CLARK;*MR. COCHRAN; MR. SIMPSON; NICOLE BROWN;*MR. DARDEN; OPENING STATEMENT; LOS ANGELES; MR. COCHRAN; DETECTIVE FUHRMAN; DISCUSSION HELD; WOULD

LIKE;*MR. DOUGLAS; BROWN SIMPSON; THANK YOU.; MR. DARDEN; DEPUTY DISTRICT; FOLLOWING PROCEEDINGS; DISTRICT ATTORNEYS.; MISS CLARK

Enter Selection (0-10)>> 6 Enter Search Word: MURDER MURDER occurred: 125 times

Enter Selection (0-10)>> 7
Enter word to Concord: KILL
Building index...
Displaying 15 of 84 matches:
WAS IN EMINENT DANGER AND NEEDED TO KILL IN SELF-DEFENSE. BUT THE ARIS COURT
R OCCURRED."I KNOW HE'S GOING TO KILL ME. I WISH HE WOULD HURRY UP AND GET FLICTED HARM WAY BEYOND NECESSARY TO KILL THE VICTIM. AND THIS IS A QUOTE FROM
'M GOING TO HURT YOU , I'M GOING TO KILL YOU , I'M GOING TO BEAT YOU."THO HAVE HER AND NO ONE ELSE WILL IS TO KILL HER. THAT IS CLEAR IN THE RESEARCH. ELLED OUT TO HIM ,"HE'S GOING TO KILL ME."IT WAS CLEAR TO OFFICER EDWAR NNING OUT SAYING,"HE'S GOING TO KILL ME ,"THEN THE DEFENDANT ARRIVES I NS OUT AND SAYS ,"HE'S TRYING TO KILL ME."SHE'S LITERALLY IN FLIGHT. S HER DURING THE BEATING THAT HE WOULD KILL HER,AND THE DEFENDANT CONTINUED TH TATEMENT OF THE DEFENDANT,"I'LL KILL YOU ,"CONSTITUTES COMPOUND HEARSA FFICER SCREAMING ,"HE'S GOING TO KILL ME."ICAN'T IMAGINE A STRONGER C "HE'S GOING CRAZY. HE IS GOING TO KILL ME."THIS IS VERY SIMILAR TO THE S NNER THAT SHE BELIEVED THAT HE WOULD KILL HER. NOW , MR. UELMEN HAS TALKED ABO
OF A DOMESTIC VIOLENCE,A MOTIVE TO KILL,THE FINAL ACT OF CONTROL.THERE IS CTIM THAT THE DEFENDANT HAD TRIED TO KILL HER PREVIOUSLY WAS USED TO SHOW THAT

Enter Selection (0-10)>> 8
Enter seed word: MURDER
Building word-context index...
court and case evidence defendant time jury crime motion relationship statement witness issue so that trial blood defense person problem

```
Enter Selection (0-10) >> 9
Find first occurrence of what Word? : GLOVE First Occurrence of: GLOVE is at offset: 93811

Enter Selection (0-10) >> 10
Compiling Vocabulary Frequencies Building vocabulary index... [('THE', 19386), (',', 18677), ('TO', 11634), ('THAT', 10777), ('AND', 8938), (':', 8369), ('OF', 8198), ('*', 7850), ('IS', 6322), ('I', 6244), ('A', 5590), ('IN', 5456), ('YOU', 4879), ('WE', 4385), ('THIS', 4264), ('IT', 3815), ('COURT', 3763), ('WAS', 3255), ('HAVE', 2816), ('–', 2797), ('?', 2738),
.
. Edited for brevity
.
('WORLDWIDE', 1), ('WORRYING', 1), ('WORSE.', 1), ('WORTHWHILE', 1), ('WOULDBE', 1), ('WOULDN'T', 1), ('WOUNDS.', 1), ('WRECKED.', 1), ('WRENCHING', 1), ('WRESTLING.', 1), ('WRISTS.', 1), ('WRITERS', 1), ('WRONGLY', 1), ('X-RAYS', 1), ('XANAX', 1), ('XEROXED.', 1), ('XEROXING.', 1), ('YAMAUCHI.', 1), ('YARD.', 1), ('YARDS', 1), ('YEAGEN', 1), ('YELL', 1), ('YELLING.', 1), ('YEP.', 1), ('YIELD', 1), ('YOUNGSTERS', 1), ('YOURS.', 1), ('YUDOWITZ', 1), ('YUDOWITZ.', 1), ('Z', 1), ('ZEIGLER', 1), ('ZERO', 1), ('ZLOMSOWITCH'S', 1)]
```

Now that you have a working example of NLTK, I suggest that you experiment with the application and start to add new NLTK operations to the classNLTKQuery in order to further explore the possibilities of Natural Language experimentation.

CHAPTER REVIEW

In this chapter, I introduced the concept of NLP in order to expand your thinking beyond simple acquire, format, and display forensic applications. I discussed some of the history of NLP and Alan Turing’s intriguing question. I then introduced you to the NLTK that performs much of the heavy lifting and allows us to experiment with NLP in Python almost immediately. I also introduced you to the concept and application of a text-based corpus and used a small sample from the transcripts of the O.J. Simpson Trial to create a small corpus for experimentation. I walked through some of the basic NLTK methods and operations in detail in order to not only create but also query our corpus. Once comfortable with the basics, I created the NLTKQuery application that

abstracted the NLTK functions through a menu driven interface. The heart of NLTKQuery is the classNLTKQuery. This class can easily be extended to perform more complex operations and dive deeper into NLTK and I challenge you to do so.

SUMMARY QUESTIONS

1. Expand the NLTKQuery class by adding the following capabilities: a. Create a new class method that generates a Frequency Distribution Graph for the Simpson Trial showing how the language in the transcripts shifts throughout the trial. (Hint: Experiment with the dispersion_plot method that can be applied to the self.textCorpus object.)

b. Create a new method that will generate a list of unusual words. This may be words that are unusually long or are not found in the standards dictionary. (Hint: Download the NLTK Data found at http://nltk.org/data.html and leverage the "word list" corpus to filter out common words.)

c. Create a new method that can identify names or places. (Hint: Leverage the "names" corpus match names found in the Simpson Corpus.) 2. From your own reference, collect a set of text files and create your own corpus for a domain or genre that will help the forensic field.
Additional Resources 203
Additional Resources
Project Gutenberg. http://www.gutenberg.org/.

Harris MD. Introduction to Natural Language Processing. Reston, VA: A Prentice-Hall Company: Reston Publishing Company, Inc.; 1985.
Turing A. Computing machinery and intelligence. http://www.csee.umbc.edu/courses/471/ papers/turing.pdf; October 1950.

CHAPTER

Network Forensics: Part I 8

CHAPTER CONTENTS

Network Investigation Basics
..... 205

What are these Sockets? 206
The Simplest Network Client Server Connect Using Sockets 208
server.py Code 208
client.py Code 209
Server.py and Client.py Program Execution 210
Captain Ramius: Re-verify Our Range to Target.. . One Ping Only
..... 211
wxPython 212
ping.py 213
guiPing.py code.....218
Ping Sweep execution 224
Port Scanning .....
225
Examples of Well-Known Ports 226
Examples of Registered Ports 226
Chapter Review .....
235
Summary Questions .....
235
Additional Resources .....
235

NETWORK INVESTIGATION BASICS

Investigating modern network environments can be fraught with difficulties. This is true whether you are responding to a breach, investigating insider activities, performing vulnerability assessments, monitoring network traffic, or validating regulatory compliance.

Many professional tools and technologies exist from major vendors like McAfee, Symantec, IBM, Saint, Tenable, and many others. However, a deep understanding of what they do, how they do it, and whether the investigative value is complete can be somewhat of a mystery. There are also free tools like Wireshark that perform network packet capture and analysis.

In order to uncloak some of the underpinnings of these technologies, I will examine the basics of network investigation methods. I will be leveraging the Python Standard Library, along with a couple of third-party libraries to accomplish the cookbook examples. I will be walking through the examples in

considerable detail, so if this is your first interaction with network programming you will have sufficient detail to expand upon the examples.

© 2014 Elsevier Inc. All rights reserved.

What are these sockets?
When interacting with a network, sockets are the fundamental building block allowing us to leverage the underlying operating system capabilities to interface with the network. Sockets provide an information channel for communicating between network endpoints, for example, between a client and server. You can think about

sockets as the endpoint of the connection between a client and a server. Applications developed in languages like Python, Java, Cþþ, and C# interface with network sockets utilizing an application programming interface (API). The sockets API on most systems today is based upon the Berkeley sockets. Berkeley sockets were originally provided with UNIX BSD Version 4.2 back in 1983. Later around 1990, Berkeley released a license-free version that is the basis of today's socket API across most operating systems (Linux, Mac OS, and Windows). This standardization provides consistency in implementation across platforms.

Figure 8.1 depicts a sample network where multiple hosts (endpoints) are connected to a network hub. Each host has a unique Internet Protocol (IP) address, and for this simple network we see that each host has a unique IP address.

These IP addresses are the most common that you will see in local area network setting. These specific addresses are based on the Internet Protocol Version 4 (IPv4) standard and represent a Class C network address. The Class C address is commonly written in a dotted notation such as 192.168.0.1. Breaking the address down into the component parts, the first three octets or the first 24 bits are considered the network address (aka the Network Identifier, or NETID). The fourth and final octet or 8 bits are considered the Local Host Address (aka the Host Identifier, or HOSTID).

192.168.0.5 192.168.0.86

192.168.0.255

192.168.0.22

Socket Connecon FIGURE 8.1
Simplest local area network.

192 .168 .0 .1
Network Address Local Host Address

In this example each host, network device, router, firewall, etc., on the local network would have the same network address portion of the IP address (192.168.0), but each will have a unique host address ranging from 0 to 255. This allows for 256 unique IP addresses within the local environment. Thus the range would be: 192.168.0.0-192.168.0.255. However, only 254 addresses are usable, this is because 192.168.0.0 is the network address and cannot be assigned to a local host, and 192.168.0.255 is dedicated as the broadcast address.

Based on this, I could use a few simple built-in Python language capabilities to create a list of IP addresses that represent the complete range. These language capabilities include a String, a List, the range function, and a “for loop.”

Specify the Base Network Address (the first 3 octets) ipBase¼'192.168.0.'

Next Create an Empty List that will hold the completed # List of IP Addresses ipList¼ []
Finally, loop through the possible list of local host # addresses 0-255 using the range function
Then append each complete address to the ipList # Notice that I use the str(ip) function in order # concatenate the string ipBase with list of numbers 0-255

```
for ip in range(0,256):
ipList.append(ipBase+str(ip)) print ipList.pop()
```

Program Output Abbreviated
192.168.0.0
192.168.0.1
192.168.0.2
192.168.0.3
..... skipped items
192.168.0.252
192.168.0.253
192.168.0.254
192.168.0.255
As you can see, manipulating IP addresses with standard Python language elements is straightforward. I will employ this technique in the Ping Sweep section later in this chapter.

127.0.0.1 Port 5555
FIGURE 8.2
Isolated localhost loopback.

The simplest network client server connect using sockets
As a way of an introduction to the sockets API provided by Python, I will create a simple network server and client. To do this I will use the same host (in other words the client and server will use the same IP address executing on the same machine), I will specifically use the special purpose and reserved localhost loopback IP address 127.0.0.1. This standard loopback IP is the same on virtually all systems and any messages sent to 127.0.0.1 never reach the outside world, and instead are automatically returned to the localhost. As you begin to experiment with network programming, use 127.0.0.1 as your IP address of

choice until you perfect your code and are ready to operate on a real network (Figure 8.2).

In order to accomplish this, I will actually create two Python programs: (1) server. py and (2) client.py. In order to make this work, the two applications must agree on a port that will be used to support the communication channel. (We already have decided to use the localhost loopback IP address 127.0.0.1.) Port numbers range between 0 and 65,535 (basically, any unsigned 16-bit integer value). You should stay away from lower numbered ports <1024 as they are assigned to standard network services (actually the registered ports now range as high as 49,500 but none of those are on my current system). For this application I will use port 5555 as it is easy to remember. Now that I have defined the IP address and port number, I have all the information that I need to make a connection.

IP Address and Port: One way to think about this in more physical terms. Think of the IP Address as the street address of a post office and the Port as the specific post-office box within the post office that I wish to address.

server.py code

```
#
# Server Objective
# 1) Setup a Simple listening Socket # 2) Wait for a connection request # 3) Accept a connection on port 5555
# 4) Upon a successful connection send a message to the client #

import socket # Standard Library Socket Module # Create Socket
myServerSocket ¼ socket.socket()
# Get my local host address

localHost ¼ socket.gethostname() # Specify a local Port to accept connections on
localPort ¼ 5555

# Bind myServerSocket to localHost and the specified Port # Note the bind call requires one parameter, but that # parameter is a tuple (notice the parenthesis usage)

myServerSocket.bind((localHost, localPort))
```

Begin Listening for connections myServerSocket.listen(1)

Wait for a connection request
Note this is a synchronous Call
meaning the program will halt until
a connection is received.
Once a connection is received
we will accept the connection and obtain the
ipAddress of the connector

print'Python-Forensics .... Waiting for Connection Request' conn, clientInfo ¼ myServerSocket.accept()
Print a message to indicate we have received a connection
print'Connection Received From:', clientInfo

Send a message to connector using the connection object'conn' # that was returned from the myServerSocket.accept() call # Include the client IP Address and Port used in the response

conn.send('Connection Confirmed:'+'IP:'+ clientInfo[0] +'Port:'+ str (clientInfo[1]))
client.py code
Next, the client code that will make a connection to the server

#
Client Objective
1) Setup a Client Socket
2) Attempt a connection to the server on port 5555 # 3) Wait for a reply
4) Print out the message received from the server

import socket # Standard Library Socket Module
MAX_BUFFER ¼ 1024 # Set the maximum size to receive
Create a Socket
myClientSocket ¼ socket.socket()
Get my local host address
localHost ¼ socket.gethostname()
Specify a local Port to attempt a connection
localPort ¼ 5555
Attempt a connection to my localHost and localPort

myClientSocket.connect((localHost, localPort))

Wait for a reply
This is a synchronous call, meaning
that the program will halt until a response is received # or the program is terminated

msg ¼ myClientSocket.recv(MAX_BUFFER) print msg
# Close the Socket, this will terminate the connection
myClientSocket.close()

server.py and client.py program execution
Figure 8.3 depicts the program execution. I created two terminal windows, the top is the execution of server.py (which I started first) and the bottom is the execution of client.py. Notice that the client communicated from the source port 59,714, this was chosen by the socket service and not specified in the client code. The server port 5555 in this example is the destination port.

I realize this does not provide any investigative value, however it does provide a good foundational understanding of how network sockets function and this is a prerequisite to understanding some of the probative or investigative programs.

FIGURE 8.3
server.py/client.py program execution.
CAPTAIN RAMIUS: RE-VERIFY OUR RANGE TO TARGET... ONE PING ONLY

You probably remember this famous line from the book and then the movie “The

Hunt for Red October" [CLANCY] spoken so eloquently by Sean Connery as the character Marko Ramius. Of course they were using a sonar wave to calculate the distance between the Red October and the USS Dallas (Figure 8.4).

Similar to submarine warfare, one of the key elements to network investigation is the discovery of all the hosts (or more generally referred to as endpoints) on a network. This is accomplished by sending a ping (using an Internet Control Message Protocol, or ICMP for short) to each possible IP address on a network. The IP addresses that respond provide us with two vital pieces of information: (1) if they respond we know they are there and responsive and (2) how long it took for the response to be returned. One special note, many modern firewalls block ICMP messages as these can be used by hackers to perform reconnaissance activities on networks. This is also true of modern operating systems, by default they will not respond to ICMP. However, inside the network they provide a valuable service to locate and detect endpoints on a network.

For this next cookbook example, I will develop a ping sweep application in Python to scan a local network for available IP addresses. I will be using a couple special modules for this section. First I will be using wxPython to build a simple graphical user interface (GUI) for the ping sweep application. Second, I will be using a third-party module, Ping.py which is completely written in Python that handles the heavy lifting of the ICMP protocol.

I choose to develop the applications in this chapter within a GUI environment for two reasons—first to give you exposure to a cross platform GUI environment wxPython, and second because the execution of ping sweeps by using command line options would be quite tedious and the GUI interface will simplify the interaction.

FIGURE 8.4
Photo of the actual USS Dallas Los Angeles-class nuclear-powered attack submarine.

wxPython

As you have seen throughout this book one of the advantages of Python is the out-ofthe-box cross-platform capabilities that it provides. In the spirit of Python, wxPython provides GUI capabilities that are also cross platform (Windows, Linux, and Mac). This library allows us to build fully functional GUI-based applications that integrate directly in the standard Python language and structure. The simple GUI applications in this chapter are only a brief introduction to wxPython, as I am trying to keep the first GUI application as simple and easy to understand as possible. As I move forward I will be utilizing wxPython throughout the remaining chapters in the book.

To obtain more information about wxPython and to install the environment, visit the http://www.wxPython.org/ project page. Third-party libraries like wxPython have multiple versions that can support different versions of Python and different operating systems (i.e., Windows, Mac, and Linux). Make sure that you choose the installation that is compatible with your configuration.

ping.py
The ping.py module is available at http://www.g-loaded.eu/2009/10/30/Pythonping/ and is an open source Python module that handles the details of ICMP operations completely written in Python. Since this is an open source module, I am including the source here for your inspection, and including all the appropriate attributions and revisions.

```
[PYTHON PING]
#!/usr/bin/env Python
"""
A pure Python ping implementation using raw socket.
Note that ICMP messages can only be sent from processes running as root.

Derived from ping.c distributed in Linux's netkit. That code is copyright (c) 1989
by The Regents of the University of California. That code is in turn derived from
code written by Mike Muuss of the US Army Ballistic Research Laboratory in
December, 1983 and placed in the public domain. They have my thanks.

Bugs are naturally mine. I'd be glad to hear about them. There are certainly word
- size dependencies here.

Copyright (c) Matthew Dixon Cowles, <http://www.visi.com/mdc/>.
Distributable under the terms of the GNU General Public License version 2.
Provided with no warranties of any sort.

Original Version from Matthew Dixon Cowles:
-> ftp://ftp.visi.com/users/mdc/ping.py
Rewrite by Jens Diemer:
-> http://www.Python-forum.de/post-69122.html#69122
Rewrite by George Notaras:
-> http://www.g-loaded.eu/2009/10/30/Python-ping/
Revision history

November 8, 2009
```

Improved compatibility with GNU/Linux systems.

Fixes by:
* George Notaras - http://www.g-loaded.eu Reported by:
* Chris Hallman - http://cdhallman.blogspot.com

Changes in this release:
- Re-use time.time() instead of time.clock(). The 2007 implementation worked only under Microsoft Windows. Failed on GNU/Linux. time.clock() behaves differently under the two OSes[1].

[1] http://docs.Python.org/library/time.html#time.clock

May 30, 2007
---
little rewrite by Jens Diemer:
- change socket asterisk import to a normal import
- replace time.time() with time.clock()
- delete "return None" (or change to "return" only)
- in checksum() rename "str" to "source_string"

November 22, 1997

Initial hack. Doesn't do much, but rather than try to guess what features I (or others) will want in the future, I've only put in what I need now.

December 16, 1997

For some reason, the checksum bytes are in the wrong order when this is run under Solaris 2.X for SPARC but it works right under Linux x86. Since I don't know just what's wrong, I'll swap the bytes always and then do an htons().

December 4, 2000

Changed the struct.pack() calls to pack the checksum and ID as unsigned. My thanks to Jerome Poincheval for the fix.

Last commit info:

$LastChangedDate: $ $Rev: $
$Author: $

"""
import os, sys, socket, struct, select, time
# From /usr/include/linux/icmp.h; your mileage may vary.
ICMP_ECHO_REQUEST ¼ 8 # Seems to be the same on Solaris.

def checksum(source_string): """
I'm not too confident that this is right but testing seems to suggest that it gives
the same answers as in_cksum in ping.c """
sum ¼ 0
countTo ¼ (len(source_string)/2)*2
count¼ 0
while count<countTo:

thisVal ¼ ord(source_string[count + 1])*256 + ord (source_string[count])
sum ¼ sum + thisVal
sum ¼ sum & 0xffffffff # Necessary?
count ¼ count + 2

if countTo<len(source_string):
sum ¼ sum + ord(source_string[len(source_string) - 1]) sum ¼ sum & 0xffffffff #
Necessary?

sum ¼ (sum >> 16) + (sum & 0xffff) sum ¼ sum + (sum >> 16)
answer¼sum
answer¼ answer & 0xffff

Swap bytes. Bugger me if I know why. answer¼ answer>> 8 | (answer << 8 &
0xff00)
return answer

def receive_one_ping(my_socket, ID, timeout):
"""
receive the ping from the socket.
"""
timeLeft ¼ timeout

```
while True:

startedSelect ¼ time.time()
whatReady ¼ select.select([my_socket], [], [], timeLeft) howLongInSelect ¼
(time.time() - startedSelect) if whatReady[0] ¼¼ []: # Timeout

return

timeReceived ¼ time.time()
recPacket, addr ¼ my_socket.recvfrom(1024)
icmpHeader ¼ recPacket[20:28]
type, code, checksum, packetID, sequence ¼ struct.unpack(

"bbHHh", icmpHeader
)
if packetID ¼¼ ID:

bytesInDouble ¼ struct.calcsize("d")
timeSent ¼ struct.unpack("d", recPacket[28:28 + bytesInDouble])[0]
return timeReceived - timeSent

timeLeft ¼ timeLeft - howLongInSelect
if timeLeft <¼ 0:
return

def send_one_ping(my_socket, dest_addr, ID): """
Send one ping to the given>dest_addr<. """
dest_addr ¼ socket.gethostbyname(dest_addr)

# Header is type (8), code (8), checksum (16), id (16), sequence (16)
my_checksum ¼ 0

# Make a dummy header with a 0 checksum.
header¼ struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, my_checksum, ID,
1)
bytesInDouble ¼ struct.calcsize("d")
data ¼ (192 - bytesInDouble) * "Q"
data ¼ struct.pack("d", time.time()) + data
```

Calculate the checksum on the data and the dummy header. my_checksum ¼ checksum(header + data)

Now that we have the right checksum, we put that in. It's just easier # to make up a new header than to stuff it into the dummy. header¼ struct.pack(

"bbHHh", ICMP_ECHO_REQUEST, 0, socket.htons(my_checksum), ID, 1)
packet¼ header + data
my_socket.sendto(packet, (dest_addr, 1))#Don't know about the 1

def do_one(dest_addr, timeout):
"""
Returns either the delay (in seconds) or none on timeout. """
icmp ¼ socket.getprotobyname("icmp")
try:

my_socket ¼ socket.socket(socket.AF_INET, socket.SOCK_RAW, icmp)
except socket.error, (errno, msg):

if errno¼¼ 1:
# Operation not permitted
msg ¼ msg + (

" - Note that ICMP messages can only be sent from processes"

" running as root.")
raise socket.error(msg)

raise # raise the original error
my_ID¼ os.getpid() & 0xFFFF
send_one_ping(my_socket, dest_addr, my_ID) delay¼
receive_one_ping(my_socket, my_ID, timeout)
my_socket.close()
return delay
def verbose_ping(dest_addr, timeout ¼ 2, count ¼ 4):

"""
Send>count< ping to >dest_addr< with the given>timeout< and display the result.

```
"""
for i in xrange(count):

print "ping %s..." % dest_addr,
try:
delay ¼ do_one(dest_addr, timeout)

except socket.gaierror, e:
print "failed. (socket error:'%s')" % e[1] break

if delay¼¼ None:
print "failed. (timeout within %ssec.)" % timeout
else:
delay ¼ delay * 1000
print "get ping in %0.4fms" % delay
print
if __name__¼¼'__main__':
verbose_ping("heise.de")
verbose_ping("google.com")
verbose_ping("a-test-url-taht-is-not-available.com")
verbose_ping("192.168.1.1")
```

I have provided detailed documentation in line with the program so you can walk through the program reading the comments for clarity. Figures 8.5 and 8.6 depict the launch and startup GUI of Ping Sweep. Notice in Figure 8.5 that I launched the program from the command line with administrative privilege. This is necessary as administrator privilege is required to perform the ping operations. Before examining the code, take a look at the overall layout of the program. I recommend that you start by examining the “Setup the Application Windows” section a couple pages down in the code. Then, I would move back to the beginning of the code and examine the pingScan event handler starting with “def pingScan(event)”:

FIGURE 8.5
Command line launch of the guiPing.py as root.

FIGURE 8.6
GUI interface for Ping Sweep.
I chose a simple GUI design with just two buttons, Scan and Exit, along with several spin controls to specify the base IP address and Local Host Range.
guiPing.py code

```
#
# Python Ping Sweep GUI Application
#

import wxversion # Specify the proper version of wxPython
wxversion.select("2.8")
# Import the necessary modules
import wx # Import the GUI module wx
import sys # Import the standard library module sys import ping # Import the ICMP Ping Module
import socket # Import the standard library module socket

from time import gmtime, strftime # import time functions
```

```
#
# Event Handler for the pingScan Button Press
# This is executed each time the Scan Button is pressed on the GUI #

def pingScan(event):

# Since the user specifies a range of Hosts to Scan, I need to verify # that the
startHost value is <¼ endHost value before scanning # this would indicate a
valid range
# If not I need to communicate the error with the user

if hostEnd.GetValue() < hostStart.GetValue():
# This is an improper setting
# Notify the user using a wx.MessageDialog Box
dlg ¼ wx.MessageDialog(mainWin,"Invalid Local Host Selection","Confirm",
wx.OK | wx.ICON_EXCLAMATION)

result ¼ dlg.ShowModal() dlg.Destroy()
return

# If we have a valid range update the Status Bar
mainWin.StatusBar.SetStatusText('Executing Ping Sweep .... Please Wait'
# Record the Start Time and Update the results window

utcStart ¼ gmtime()
utc ¼ strftime("%a, %d %b %Y %X +0000", utcStart)
results.AppendText("\n\nPing Sweep Started: "+ utc+ "\n\n")

# Similar to the example script at the beginning of the chapter # I need to build
the base IP Address String
# Extract data from the ip Range and host name user selections # Build a Python
List of IP Addresses to Sweep
baseIP¼ str(ipaRange.GetValue())+'.'+str(ipbRange.GetValue())+'.'
+str(ipcRange.GetValue())+'.'

ipRange ¼ []
for i in range(hostStart.GetValue(), (hostEnd.GetValue()+1)):
ipRange.append(baseIP+str(i))
# For each of the IP Addresses in the ipRange List, Attempt an PING
for ipAddress in ipRange:
```

```
try:
# Report the IP Address to the Window Status Bar # Prior to the attempt
mainWin.StatusBar.SetStatusText('Pinging IP:'+ ipAddress)
# Perform the Ping
delay ¼ ping.do_one(ipAddress, timeout¼2)
# Display the IP Address in the Main Window
results.AppendText(ipAddress+'\t')

if delay ! ¼ None:
# If Successful (i.e. no timeout) display # the result and response time

results.AppendText('Response Success')
results.AppendText('Response Time:'+str(delay)+' Seconds')
results.AppendText("\n")

else :
# If delay¼¼ None, then the request timed out # Report the Response Timeout
results.AppendText('Response Timeout')
results.AppendText("\n")

except socket.error, e:
# If any socket Errors occur Report the offending IP # along with any error
information provided by the socket

results.AppendText(ipAddress)
results.AppendText('Response Failed:') results.AppendText(e.message)
results.AppendText("\n")

# Once all ipAddresses are processed
# Record and display the ending time of the sweep

utcEnd¼ gmtime()
utc ¼ strftime("%a, %d %b %Y %X +0000", utcEnd)
results.AppendText("\nPing Sweep Ended: "+ utc + "\n\n")

# Clear the Status Bar
mainWin.StatusBar.SetStatusText(")
return
# End Scan Event Handler ¼¼¼
```

#
Program Exit Event Handler
This is executed when the user presses the exit button # The program is terminated using the sys.exit() method

def programExit(event): sys.exit()
End Program Exit Event Handler ¼¼¼¼¼¼¼¼¼¼¼¼¼¼¼¼¼¼¼¼

#
Setup the Application Windows ¼¼¼¼¼¼¼¼¼¼¼¼¼¼¼¼¼¼¼¼
This section of code sets up the GUI environment

Instantiate a wx.App()objet app ¼ wx.App()
define the main window including the size and title
mainWin ¼ wx.Frame(None, title¼"Simple Ping (ICMP) Sweeper 1.0", size ¼(1000,600))
define the action panel, this is the area where the buttons and spinners # are located

panelAction ¼ wx.Panel(mainWin) # define action buttons
I'm creating two buttons, one for Scan and one for Exit
Notice that each button contains the name of the function that will # handle the button press event -- pingScan and ProgramExit respectively

scanButton ¼ wx.Button(panelAction, label¼'Scan')
scanButton.Bind(wx.EVT_BUTTON, pingScan)
exitButton ¼ wx.Button(panelAction, label¼'Exit')
exitButton.Bind(wx.EVT_BUTTON, programExit)
# define a Text Area where I can display results
Results ¼ wx.TextCtrl(panelAction, style ¼ wx.TE_MULTILINE | wx. HSCROLL)

Base Network for Class C IP Addresses have 3 components
For class C addresses, the first 3 octets (24 bits) define the network # e.g., 127.0.0
the last octet (8 bits) defines the host i.e., 0-255
Thus I setup 3 spin controls one for each of the 3 network octets # I also set the default value to 127.0.0 for convenience

ipaRange ¼ wx.SpinCtrl(panelAction, -1,") ipaRange.SetRange(0, 255)
ipaRange.SetValue(127)

ipbRange ¼ wx.SpinCtrl(panelAction, -1,") ipbRange.SetRange(0, 255)
ipbRange.SetValue(0)

ipcRange ¼ wx.SpinCtrl(panelAction, -1,") ipcRange.SetRange(0, 255)
ipcRange.SetValue(0)

Also, I'm adding a label for the user
ipLabel ¼ wx.StaticText(panelAction, label¼"IP Base: ")
Next, I want to provide the user with the ability to set the host range # they wish to scan. Range is 0 - 255

hostStart ¼ wx.SpinCtrl(panelAction, -1,") hostStart.SetRange(0, 255)
hostStart.SetValue(1)

hostEnd ¼ wx.SpinCtrl(panelAction, -1,") hostEnd.SetRange(0, 255)
hostEnd.SetValue(10)

HostStartLabel HostEndLabel ¼ wx.StaticText(panelAction, label¼"Host Start: ") ¼ wx.StaticText(panelAction, label¼"Host End: ")

Now I create BoxSizer to automatically align the different components # neatly within the panel
First, I create a horizontal Box
I'm adding the buttons, ip Range and Host Spin Controls

actionBox ¼ wx.BoxSizer()
actionBox.Add(scanButton, proportion¼1, flag¼wx.LEFT, border¼5)
actionBox.Add(exitButton, proportion¼0, flag¼wx.LEFT, border¼5)

actionBox.Add(ipLabel, proportion¼0, flag¼wx.LEFT, border¼5)

actionBox.Add(ipaRange, proportion¼0, flag¼wx.LEFT, border¼5)
actionBox.Add(ipbRange, proportion¼0, flag¼wx.LEFT, border¼5)
actionBox.Add(ipcRange, proportion¼0, flag¼wx.LEFT, border¼5)

actionBox.Add(HostStartLabel, proportion ¼0, flag¼wx.LEFT|wx.CENTER,

```
border¼5)
actionBox.Add(hostStart, proportion¼0, flag¼wx.LEFT, border¼5)

actionBox.Add(HostEndLabel, proportion¼0, flag¼wx.LEFT|wx.CENTER,
border¼5)
actionBox.Add(hostEnd, proportion¼0, flag¼wx.LEFT, border¼5)

# Next I create a Vertical Box that I place the Horizontal Box Inside # Along
with the results text area

vertBox ¼ wx.BoxSizer(wx.VERTICAL)
vertBox.Add(actionBox, proportion¼0, flag¼wx.EXPAND | wx.ALL,
border¼5) vertBox.Add(results, proportion¼1, flag¼wx.EXPAND | wx.LEFT |
wx. BOTTOM | wx.RIGHT, border¼5)

# I'm adding a status bar to the main windows to display status messages
mainWin.CreateStatusBar()
# Finally, I use the SetSizer function to automatically size the windows based on
the definitions above
panelAction.SetSizer(vertBox) # Display the main window
mainWin.Show()
# Enter the Applications Main Loop # Awaiting User Actions
app.MainLoop()
```

Ping Sweep execution

Figure 8.7 provides a summary of two executions of the Ping Sweep program. In the first run, the base IP address 127.0.0. is utilized and hosts 1-5 are selected and the results are displayed. In the second run I chose my local network 192.168.0. base address and I scanned hosts 1-7 and the results of each ping is recorded. For both runs when any host responds, the time (or delay) is also reported.

In Figure 8.8, I purposely misconfigured the Host Selection to be invalid (the starting Host is greater than the ending Host number). As expected the dialog box with the error is reported. You can examine the code that displays this dialog box in the pingScan event handler.

FIGURE 8.7
Ping Sweep execution.

FIGURE 8.8
Error handling for misconfigured host range.

As you can see most of the work is related to setting up the GUI application, and creating the list of IP addresses to scan. Once that is completed, the code that leverages the ping.py module to perform the ping and retrieve the results is only one line of code. Send one and only one ping.

Perform the Ping

delay ¼ ping.do_one(ipAddress, timeout¼2)

One of the things to keep in mind when performing a Ping Sweep to identify endpoints is to run the scan often as endpoints that are unavailable, are shutdown or have failed may not be available the first time you scan. I will provide you with a challenge problem in the summary questions to improve this application to cover this issue.

PORT SCANNING

Once I have identified endpoints within our network, the next step is to perform a port scan. What exactly is a port scan, or more specifically a TCP/IP port scan? Computers that support communication protocols utilize ports in order to make connections to other parties. In order to support different conversations with multiple parties, ports are used to distinguish various communications. For example, web servers can use the Hypertext Transfer Protocol (HTTP) to provide access to a web page which utilizes TCP port number 80 by default. The Simple Mail Transfer Protocol or SMTP uses port 25 to send or transmit mail messages. For each unique IP address, a protocol port number is identified by a 16-bit number, commonly known as the port number 0-65,535. The combination of a port number and IP address provides a complete address for communication. The parties that are communicating will each have an IP address and port number. Depending on the direction of the communication both a source and destination address (IP address and port combination) are required.

Ports are divided into three basic categories as in Table 8.1.
Table 8.1 Categories of Network Ports Category Port Range Usage

Well0-1023 known
ports
Registered 1024-49,151 ports

Dynamic 49,152-65,535 ports
These ports are used by system processes that provide network services that are widely used

Registration is managed by the Internet Corporation for Assigned Names and Numbers (ICANN). Or, more specifically, by Internet Assigned Numbers Authority (IANA) which is now operated by ICANN. [ICANN]

The ports are typically ephemeral in nature (or shortlived). The ports are allocated automatically from a predefined range by the operating system as needed. On servers these ports are used to continue communication connections with clients that originally connected to well-known ports such as the File Transfer Protocol or FTP

Examples of well-known ports
Some well-known ports that you may be familiar with are given in Table 8.2 (note that this is just a sample of the list).

Examples of registered ports
A short example of registered ports that you may be familiar with are in Table 8.3 (Note that this is just a sample of the list.)

To develop the simplest Port Scanner in Python, I need to know just a few things:

What IP address to target? (1)
(2) What port range should I scan?
(3) Whether I should display all the results or should I only display the

ports that were found to be open. In other words ports that I could successfully connect to.

Figure 8.9 depicts the GUI for our simple Port Scanner. The GUI allows the user to specify the IP address to scan along with the port range. The GUI also includes a checkbox that allows the user to specify whether all the results or only the successful results are displayed.

I have provided detailed documentation in line with the program so you can walk through the program reading the comments for clarity. Figure 8.10 depicts the launch of the startup Port Scanner GUI. As you can see in Figure 8.10, I launched the program from the command line with administrative privilege. This is necessary as administrator privilege is required to perform the port scan network operations.

Table 8.2 Examples of Well-known Ports

Port Service Name Number Transport

Protocol Description

echo 7
echo 7
ftp 21
ftp 21
ssh 22
tcp udp tcp udp tcp

ssh 22 udp

telnet 23
telnet 23
smtp 25
smtp 25
nameserver 42
nameserver 42
http 80
http 80
nntp 119
tcp udp tcp udp tcp udp tcp udp tcp

nntp 119 udp

ntp 123
ntp 123
netbios-ns 137
netbios-ns 137
snmp 161
snmp 161
tcp udp tcp udp tcp udp Echo
Echo
File Transfer [Control] File Transfer [Control] The Secure Shell (SSH) Protocol
The Secure Shell (SSH) Protocol
Telnet
Telnet
Simple Mail Transfer Simple Mail Transfer Host Name Server
Host Name Server
World Wide Web HTTP World Wide Web HTTP Network News Transfer

Protocol
Network News Transfer Protocol
Network Time Protocol Network Time Protocol NETBIOS Name Service
NETBIOS Name Service SNMP
SNMP

Table 8.3 Examples of Registered Ports

Service Name Port Transport

Number Protocol Description

nlogin nlogin telnets telnets pop3s 758 tcp 758 udp 992 tcp 992 udp 995 tcp

pop3s

995 udp nlogin service
nlogin service
telnet protocol over TLS/SSL telnet protocol over TLS/SSL pop3 protocol over TLS/SSL (was spop3)
pop3 protocol over TLS/SSL (was spop3)

FIGURE 8.9
Port Scanner GUI.

FIGURE 8.10
Port Scanner program launch.

Before diving into the code take a look at the overall layout of the program. I recommend that you start by examining the “Setup the Application Windows” section a couple pages down in the code. Then, I would move back to the beginning of the code and examine the portScan event handler starting with “def portScan(event)”:

As you can see most of the work is related to setting up the GUI application and setting up the list of host ports to scan. Once that is done, the code that actually scans each port and checks the result is only a few lines as shown here.

```
# open a socket
reqSocket ¼ socket(AF_INET, SOCK_STREAM)
# Try Connecting to the specified IP, Port

response ¼ reqSocket.connect_ex((baseIP, port)) #
# Python Port Scanner #

import wxversion
wxversion.select("2.8")

import wx # Import the GUI module wx
import sys # Import the standard library module sys import ping # Import the ICMP Ping Module
from socket import * # Import the standard library module socket

from time import gmtime, strftime # import time functions

#
# Event Handler for the portScan Button Press #

def portScan(event):
# First, I need to check that the starting port is<¼ending port value
if portEnd.GetValue() < portStart.GetValue():
```

This is an improper setting # Notify the user and return
dlg ¼ wx.MessageDialog(mainWin,"Invalid Host Port Selection", "Confirm",
wx.OK | wx.ICON_EXCLAMATION)

result¼ dlg.ShowModal() dlg.Destroy()
return

# Update the Status Bar
mainWin.StatusBar.SetStatusText('Executing Port Scan .... Please Wait')
# Record the Start Time

utcStart ¼ gmtime()
utc ¼ strftime("%a, %d %b %Y %X +0000", utcStart)
results.AppendText("\n\nPort Scan Started: "+ utc+ "\n\n")

Build the base IP Address String
Extract data from the ip Range and host name user selections # Build a Python
List of IP Addresses to Sweep
baseIP¼ str(ipaRange.GetValue())+

'.'+str(ipbRange.GetValue())+ '.'+str(ipcRange.GetValue())+
'.'+str(ipdRange.GetValue())

# For the IP Addresses Specified, Scan the Ports Specified
for port in range(portStart.GetValue(), portEnd.GetValue()+1):
try:

# Report the IP Address to the Window Status Bar
mainWin.StatusBar.SetStatusText('Scanning:'+ baseIP+' Port:'+str(port))

open a socket
reqSocket ¼ socket(AF_INET, SOCK_STREAM)
Try Connecting to the specified IP, Port
response ¼ reqSocket.connect_ex((baseIP, port))
if we receive a proper response from the port # then display the results
received

if(response ¼¼ 0) :
# Display the ipAddress and Port

```
results.AppendText(baseIP+'\t'+str(port)+'\t') results.AppendText('Open')
results.AppendText("\n")

else:

# if the result failed, only display the result # when the user has selected the
"Display All" check box if displayAll.GetValue() ¼¼ True:

results.AppendText(baseIP+'\t'+str(port)+'\t') results.AppendText('Closed')
results.AppendText("\n")

# Close the socket reqSocket.close()

except socket.error, e:
# for socket Errors Report the offending IP
results.AppendText(baseIP+'\t'+str(port)+'\t') results.AppendText('Failed:')
results.AppendText(e.message)
results.AppendText("\n")
# Record and display the ending time of the sweep utcEnd¼ gmtime()
utc ¼ strftime("%a, %d %b %Y %X +0000", utcEnd)
results.AppendText("\nPort Scan Ended: "+ utc + "\n\n)"

# Clear the Status Bar
mainWin.StatusBar.SetStatusText(")
# End Scan Event Handler ¼¼¼
```

```
#define the action panel
panelAction ¼ wx.Panel(mainWin)

#define action buttons
# I'm creating two buttons, one for Scan and one for Exit
# Notice that each button contains the name of the function that will # handle the
button press event. Port Scan and ProgramExit respectively

displayAll ¼ wx.CheckBox(panelAction, -1,'Display All', (10, 10))
displayAll.SetValue(True)
scanButton ¼ wx.Button(panelAction, label¼'Scan')
scanButton.Bind(wx.EVT_BUTTON, portScan)
exitButton ¼ wx.Button(panelAction, label¼'Exit')
exitButton.Bind(wx.EVT_BUTTON, programExit) # define a Text Area where I
can display results
results ¼ wx.TextCtrl(panelAction, style ¼ wx.TE_MULTILINE | wx.
HSCROLL)

# Base Network for Class C IP Addresses has 3 components
# For class C addresses, the first 3 octets define the network i.e 127.0.0 # the last
8 bits define the host i.e. 0-255

# Thus I setup 3 spin controls one for each of the 4 network octets # I also, set
the default value to 127.0.0.0 for convenience

ipaRange ¼ wx.SpinCtrl(panelAction, -1,'') ipaRange.SetRange(0, 255)
ipaRange.SetValue(127)

ipbRange ¼ wx.SpinCtrl(panelAction, -1,'') ipbRange.SetRange(0, 255)
ipbRange.SetValue(0)

ipcRange ¼ wx.SpinCtrl(panelAction, -1,'') ipcRange.SetRange(0, 255)
ipcRange.SetValue(0)

ipdRange ¼ wx.SpinCtrl(panelAction, -1,'') ipdRange.SetRange(0, 255)
ipdRange.SetValue(1)

# Add a label for clarity
ipLabel ¼ wx.StaticText(panelAction, label¼"IP Address: ")
```

Next, I want to provide the user with the ability to set the port range # they wish to scan. Maximum is 20 - 1025

portStart ¼ wx.SpinCtrl(panelAction, -1,") portStart.SetRange(1, 1025) portStart.SetValue(1)

portEnd ¼ wx.SpinCtrl(panelAction, -1,") portEnd.SetRange(1, 1025) portEnd.SetValue(5)

PortStartLabel ¼ wx.StaticText(panelAction, label¼"Port Start: ") PortEndLabel ¼ wx.StaticText(panelAction, label¼"Port End: ")

Now I create BoxSizer to automatically align the different components neatly
First, I create a horizontal Box
I'm adding the buttons, ip Range and Host Spin Controls

actionBox ¼ wx.BoxSizer()

actionBox.Add(displayAll, proportion¼0, flag¼wx.LEFT|wx.CENTER, border¼5)
actionBox.Add(scanButton, proportion¼0, flag¼wx.LEFT, border¼5)
actionBox.Add(exitButton, proportion¼0, flag¼wx.LEFT, border¼5)

actionBox.Add(ipLabel, proportion¼0, flag¼wx.LEFT|wx.CENTER, border¼5)

actionBox.Add(ipaRange, proportion ¼0, flag¼wx.LEFT, border¼5)
actionBox.Add(ipbRange, proportion¼0, flag¼wx.LEFT, border¼5)
actionBox.Add(ipcRange, proportion¼0, flag¼wx.LEFT, border¼5)
actionBox.Add(ipdRange, proportion¼0, flag¼wx.LEFT, border¼5)

actionBox.Add(PortStartLabel, proportion¼0, flag¼wx.LEFT|wx.CENTER, border¼5)
actionBox.Add(portStart, proportion¼0, flag¼wx.LEFT, border¼5)

actionBox.Add(PortEndLabel, proportion¼0, flag¼wx.LEFT|wx.CENTER, border¼5)
actionBox.Add(portEnd, proportion¼0, flag¼wx.LEFT, border¼5)

Next I create a Vertical Box that I place the Horizontal Box components # inside along with the results text area

vertBox ¼ wx.BoxSizer(wx.VERTICAL)
vertBox.Add(actionBox, proportion¼0, flag¼wx.EXPAND | wx.ALL, border¼5) vertBox.Add(results, proportion¼1, flag¼wx.EXPAND | wx.LEFT | wx. BOTTOM | wx.RIGHT, border¼5)

# I'm adding a menu and status bar to the main window
mainWin.CreateStatusBar()
# Finally, I use the SetSizer function to automatically size the windows # based on the definitions above
panelAction.SetSizer(vertBox)
# Display the main window

mainWin.Show() # Enter the Applications Main Loop # Awaiting User Actions

app.MainLoop

Now that you have reviewed the code, Figures 8.11 and 8.12 depict program execution. The only difference between the two figures is the setting of the Display All checkbox.

FIGURE 8.11
Port Scanner execution with Display All selected.

FIGURE 8.12
Port Scanner execution with Display NOT selected.
Additional Resources 235

CHAPTER REVIEW

In this chapter, I first introduced the concept of digital investigation of a network. Then I walked through the fundamentals required to perform basic synchronous network socket operations within Python. I created three programs: (1) Server and (2) Client in order to demonstrate how connections are made. Next, I discussed the concept of a ping sweep and the value it can bring to network-based investigations. I then created a basic ping sweeper application. This application employed two third-party modules: wxPython and ping.py. wxPython was used to create a simple GUI application to control ping sweep operations. Finally, I developed a port scanning application again using a GUI, and created the application entirely in Python.

In Chapter 9 “Network Investigation Part II,” I will expand the Port Scan application to provide examples of OS fingerprinting and provide an example of passive monitoring to identify host and port usage.

SUMMARY QUESTIONS

1. What are the investigative benefits that can be obtained when performing a ping sweep?

2. What are the investigative benefits that can be obtained when performing a port scan?
3. Modify the ping sweep application to save the results of the sweep in a Python list.
4. Modify the ping sweep application and add additional scanning options. For example:
a. Program the scan to automatically and repeatedly run at predefined intervals to identify a broader range of endpoints. Keep track of the IPs identified in the list (or you might choose a set for this operation, why would a Python set be beneficial?).
b. Create a stealth mode that randomly pings the hosts within the specified address range over a multiday time period.
5. Modify the port scan application to expand the range of allowed ports.
6. Integrate the port scan and ping sweep applications in such a way that port scans will be executed automatically against hosts responding to a ping and ignore nonresponding hosts.

Additional Resources
Internet Corporation for Assigned Numbers and Names, http://www.icann.org/.
The Hunt for Red October, http://www.tomclancy.com/book_display.php?isbn13¼

9780425240335 .
Python Ping Module, https://pypi.Python.org/pypi/ping.
wxPython GUI environment, www.wxPython.org.

CHAPTER

Network Forensics: Part II 9

CHAPTER CONTENTS

Introduction 237
Packet Sniffing 238
Raw Sockets in Python 240
What is Promiscuous Mode or Monitor Mode? 240

Setting Promiscuous Mode Ubuntu 12.04 LTS Example 240
Raw Sockets in Python Under Linux 241
Unpacking Buffers 242
Python Silent Network Mapping Tool
..... 247
PSNMT Source Code .....
249
psnmt.py Source Code 249
decoder.py Source Code 253
commandParser.py 256
classLogging.py Source Code 257
csvHandler.py Source Code 258
Program Execution and Output
..... 259
Forensic Log 260
TCP Capture Example 260
UDP Capture Example 261
CSV File Output Example 261
Chapter Review .....
262
Summary Question/Challenge
..... 262
Additional Resources .....
263

INTRODUCTION

As we discovered in Chapter 8, Python has a rich set of Standard Library capabilities to perform network interface, discovery, and analysis. The Ping Sweep and Port Scan applications are quite straightforward once you become familiar with the underlying libraries and modules. However, interactive scanning and probing has several important limitations:

1. In order for the sweeps and scans to be effective, the targets (hosts, routers,

© 2014 Elsevier Inc. All rights reserved.

switches printers, servers) need to be powered on and functional in order to interact with them.

2. The environment that you are working in must be tolerant to these types of

“noisy” scanning activity. Actually, most intrusion prevention systems (IPS) are precisely looking for this type of activity and will categorize them as attacks and respond accordingly, unless they are configured to ignore them. This is more difficult to accomplish than you might think, and most cyber-security folks are not inclined to make configuration changes to their IPS.

3. Port scans are only effective when services owning these ports respond to the probes, as we would expect. Malicious services that we may be searching for do not necessarily play nice and would not respond to these novice inquiries.

4. Many of the malicious services utilize the User Datagram Protocol (UDP) when communicating with their handlers and can operate silently when being probed by vulnerability assessment technologies.

5. Finally, the operators of critical infrastructure environments are unlikely to allow active scanning and probing of those networks, because in many cases the probing activity may disrupt operations and crash systems. If this were to happen in a supervisory control and data acquisition environment, these disruptions or system crashes could shut the lights off for thousands of customers or worse.

PACKET SNIFFING

Network (or packet) sniffing is another method that can be used, and when used properly can provide insight. Network sniffing, again if done well, can provide three critical benefits over port scanning:

1. The sniffer is completely silent and will not place a single packet on the network ensuring zero impact on network operations.
2. The sniffer is an observer that can run for hours, days, weeks, months, or even continuously collecting information that will more completely describe activities of local hosts, servers, network devices, or even rogue devices that were previously undetected.
3. Finally, it can capture activities that are stealthy and only occur periodically or sporadically.

In the primitive form, a packet sniffer (also referred to a network sniffer) captures all of the packets of data that pass through a given network interface. In order to capture these packets, your network interface must be in Promiscuous Mode and the interface needs to be connected to a port that has visibility to all of the packets. For example, if you are interested in a specific subnet, you would

connect the sniffer to a switch or hub on that subnet. Most modern switches today support port mirroring via a Switched Port ANalyzer (SPAN) or Remote Switched Port ANalyzer as shown in Figures 9.1 and 9.2. These ports are typically connected to IPS, network monitoring devices or performance measurement devices that can detect network loading.

FIGURE 9.1
SPAN port diagram.

FIGURE 9.2
SPAN port connections.

SPAN ports are most commonly attributed to Cisco (where they were originally referred to as port mirroring). Modern switches can be configured to mirror specific network ports to a common interface used for network monitoring and interface with a variety of security appliances.

RAW SOCKETS IN PYTHON
In order to perform packet sniffing in Python, we need the following:

1. We must be using a network interface card (NIC) that has the ability to operate in Promiscuous Mode.
2. On most modern operating systems—i.e., Windows, Linux, and Mac OS X—you must also have administrator privilege.
3. Once we have accomplished 1þ2, we can create a raw socket.

What is Promiscuous Mode or Monitor Mode?
When a capable NIC is placed in Promiscuous Mode, it allows the NIC to intercept and read each arriving network packet in its entirety. If the NIC is not in Promiscuous

Mode, it will only receive packets that are specifically addressed to the NIC. Promiscuous Mode must be supported by the NIC and by the operating system

and any associated driver. Not all NICs support Promiscuous Mode, however it is pretty easy to determine if you have a NIC and OS capable of Promiscuous Mode.

Setting Promiscuous Mode Ubuntu 12.04 LTS Example
On Linux you can place your NIC into Promiscuous Mode by using ifconfig command. (Note administrative rights are required.)

Command to enable Promiscuous Mode:
chet@PythonForensics:$ sudo ifconfig eth0 promisc
Next I validate the result:
(Notice the message UP BROADCAST RUNNING PROMISC MULTICAST

message.)
chet@PythonForensics:$ sudo ifconfig

eth0 Link encap:Ethernet HWaddr 00:1e:8c:b7:6d:64
inet addr:192.168.0.25 Bcast:192.168.0.255 Mask:255.255.255.0 inet6 addr: fe80::21e:8cff:feb7:6d64/64 Scope:Link
UP BROADCAST RUNNING PROMISC MULTICAST MTU:1500 Metric:1
RX packets:43284 errors:0 dropped:0 overruns:0 frame:0
TX packets:11338 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:17659022 (17.6 MB) TX bytes:1824060 (1.8 MB)

Next I have turned off Promiscuous Mode and validate the result: You notice that the message now states: UP BROADCAST RUNNING MULTICAST without the PROMISC

chet@PythonForensics:$ sudo ifconfig eth0 –promisc
chet@PythonForensics:$ sudo ifconfig eth0
eth0 Link encap:Ethernet HWaddr 00:1e:8c:b7:6d:64

inet addr:192.168.0.25 Bcast:192.168.0.255 Mask:255.255.255.0 inet6 addr: fe80::21e:8cff:feb7:6d64/64 Scope:Link UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1 RX packets:43381 errors:0 dropped:0 overruns:0 frame:0 TX packets:11350 errors:0 dropped:0 overruns:0 carrier:0 collisions:0 txqueuelen:1000
RX bytes:17668285 (17.6 MB) TX bytes:1827000 (1.8 MB)

Once you have determined that you have a NIC capable of entering Promiscuous Mode, we are ready to work with raw sockets in Python.
Raw sockets in Python under Linux

Special Note: I will be using a Linux environment throughout the rest of this chapter. Since the code for handling raw sockets differs between operating systems, the code that interfaces with the raw socket will need to be modified to support Windows.

Creating a raw socket is quite straightforward in Python, as the script below demonstrates. The script accomplishes the following:

(1) Enable Promiscuous Mode on the NIC
(2) Create a raw socket
(3) Capture the next TCP packet passing by the NIC
(4) Print the contents of the packet
(5) Disable Promiscuous Mode on the NIC
(6) Close the raw socket

```
# Note: Script must be run with admin privledge

# import the socket and os libraries import socket
import os

# issue the command to place the adapter in promiscious mode
ret¼os.system("ifconfig eth0 promisc)"
# if the command was successful continue if ret ¼¼ 0:

# Create a Raw Socket in Linux
# AF_INET specifies ipv4 packets
# SOCK_RAW specifies a raw protocol at the network layer # IPPROTO_TCP
specifies the protocol to capture

mySocket¼socket.socket(socket.AF_INET, socket.SOCK_RAW,
socket.IPPROTO_TCP)

# Receive the next packet up to 255 bytes
# Note this is a synchronous call and will wait until # a packet is received

recvBuffer, addr¼mySocket.recvfrom(255)
```

```
# Print out the contents of the buffer
print recvBuffer
ret¼os.system("ifconfig eth0 -promisc)"

else:
# if the system command fails print out a message print'Promiscious Mode not Set'
```

The script will create output as shown in Figure 9.3. As you can see the output of the packet contents appears quite cryptic.

Unpacking buffers

Extracting information from a buffer like this one may seem quite tedious, as we must parse the pertinent information from the buffer. In order to deal with buffered data like this one with a defined structured, Python provides the function unpack (). Many examples and applications of unpack are available on the web that parse various well-known data structures, however the explanation of how the functions work are typically left to the reader's imagination, or at least requires further research.

For illustration, you will find examples on the web like this one to extract information from an IPv4 header.

```
ipHeader¼packet[0:20]
buffer¼unpack('!BBHHHBBH4 s4 s', ipHeader)
```

```
E\000\0004  ýℓ\000ℓACK2Å \000\000SOH \000\000SOHAPCSÇP℅CST2ÈnµRTjPADDLESOH\000þ(\000\000SOHSOHBS
\000|&ó\000|&é
```

FIGURE 9.3
Raw TCP/IP packet contents.

As with all chapters in this book, I want to make sure that you deeply understand how functions work, so that you can apply them to the challenge at hand, as well as to other problems in the future. The unpack()function takes two parameters, the first is a string that defines the format of the data held in the buffer and second is the buffer that needs to be parsed. The function returns a tuple which can be processed just like a list.

In order to figure this out, we must first study the structure of an IPv4 packet header as shown in Figure 9.4 and compare that with the format string "!BBHHHBBH4 s4 s."

Each of the characters of the format string have a specific meaning which governs the processing of the unpack() function. If you get the format string wrong, then you will get garbage back, or if the buffer you pass as the second variable does not conform to the exact format you specify, the results will be incorrect.

Let us examine the meaning of each of the characters in the format string "!BBHHHBBH4 s4 s." Note: other format specifications exist and are documented within the Standard Library documentation [UNPACK]:

Format Python Type Bytes

! Big Endian
B Integer 1
H Integer 2
s String n

IPv4 Header Format

Offsets Bytes 0 1 2 3
Octet
Bit

01 2 3 456 78 9
1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

E
0 0 Version IHL DSCP C Total Length
N

4 32 Identification Flags Fragment Offset
8 64 Time To Live **Protocol** Header Checksum
12 96 **Source IP Address**
16 128 **Destination IP Address**
20 160 Options (if IHL > 5)

FIGURE 9.4
Typical IPv4 packet header.

The first character in the format string represents the byte order of the data, for network packets this is in big endian format. This is represented as the exclamation point that is the first character in the format specification. You could also use the ">" greater than sign which also means big endian, but the explanation point is there for those of us who can never remember the typical

byte order of network data. Actually, it is good to use the exclamation point because this immediately identifies the format string as relating to a network packet.

The table below provides a mapping of each of the format characters related to the IPv4 header.
"!BBHHHBBH4 s4 s"
Size Mapping to Format (Bytes) IPv4
Definition
B 1 Version and IHL

B 1 DSCP and ECN
H 2 Total length
H 2 Identification
H 2 Flags and fragment offset
B 1 Time to live (TTL)
B 1 Protocol

H 2 Header Checksum value

4 s 4 Source IP address
4 s 4 Destination IP address 4-bit version field (this will be 4, for IPv4) 4-bit Internet Header Length representing the number of 32 bit words contained in the header
7-bit Differentiated Services Code Point 1-bit Congestion Notification
16 bits defines the entire packet size 16 bits identifier for a group of IP fragments
3-bit fragmentation flag
13-bit fragment offset value

8-bit TTL value to prevent packet looping

8-bit value identifying the protocol used in the data portion of the packet
16-bit checksum value for error detection

4-byte source IP address
4-byte destination IP address

Now that you understand the meaning of the format string and the basic unpack() function. The following code will unpack an IPv4 header and then extract each of the fields into variables for processing. I also included the code

here to convert the source and destination IP addresses into the human readable forms using the built-in socket method.

# unpack an IPv4 packet
# note the packet variable is a buffer returned from # a socket.recvfrom() method that was illustrated in Figure 9.3.
ipHeaderTuple¼unpack('!BBHHHBBH4s4s', packet)
# Field Contents

verLen ¼ipHeaderTuple[0] dscpECN ¼ipHeaderTuple[1]
packetLength¼ipHeaderTuple[2] packetID
flagFrag
timeToLive
protocol
checkSum
sourceIP
destIP
¼ipHeaderTuple[3] ¼ipHeaderTuple[4] ¼ipHeaderTuple[5] ¼ipHeaderTuple[6] ¼ipHeaderTuple[7] ¼ipHeaderTuple[8] ¼ipHeaderTuple[9] # Field 0: Version and Length # Field 1: DSCP and ECN
Field 2: Packet Length # Field 3: Identification # Field 4: Flags/Frag Offset # Field 5: Time to Live (TTL) # Field 6: Protocol Number # Field 7: Header Checksum # Field 8: Source IP
Field 9: Destination IP

Convert the sourceIP and destIP into a # standard dotted-quad string representation # for example'192.168.0.5'

sourceAddress¼socket.inet_ntoa(sourceIP); destAddress ¼socket.inet_ntoa(destIP);
# Extract the version and header size, this will give # us the offset to the data portion of the packet

version
length
ipHdrLength fragOffset fragment
¼verLen>>4 # get upper nibble version ¼verLen & 0x0F # get lower nibble header length ¼length * 4 # calculate the hdr size in bytes ¼flagFrag & 0x1FFF # get lower 13 bits... ¼fragOffset*8# calculate start of fragment

Next, we would use the same process to extract fields from the data portion of the packet, which in this case is the TCP header. You can determine the type of the data portion of the packet by examining the protocol field. Figure 9.5 depicts a typical TCP header and the format string “!HHLLBBHHH” along with the unpack() function, this can be used to extract the individual fields of the TCP header.

By using the results of the IPv4 header unpacking, we can # strip the TCP Header from the original packet

Note the ipHdrLength is the offset from the beginning of # the buffer. The standard length of a TCP packet is 20 # bytes. For our purposes these 20 bytes contain the # the pertinent information we are looking for

stripTCPHeader¼packet[ipHdrLength:ipHdrLength+ 20]

TCP Header

Offset Bytes 0 1 2 3
Octet BITS 0 1 2 3 4 5 6 7 8 9 10 111213 14151617 181920 212223 24252627 282930 31
00 Source port Destination port
4 32 Sequence number
8 64 Acknowledgment number (if ACK set)
12
96
Data offset
Reserved
N
C E U A P R S F
W C R C S S Y I Window Size0 0 0 S R E G K H T N N
16 128 Checksum Urgent pointer (if URG set)
20 160 Options (if data offset > 5. Padded at the end with "0" bytes if necessary.) ... ...

FIGURE 9.5
Typical TCP packet header.

# unpack returns a tuple, for illustration I will extract # each individual values using the unpack() function
tcpHeaderBuffer¼unpack('!HHLLBBHHH', stripTCPHeader)

sourcePort
destinationPort
sequenceNumber
acknowledgement
dataOffsetandReserve tcpHeaderLength
flags

FIN
SYN
RST
PSH
ACK
URG
ECE
CWR
windowSize
tcpChecksum
urgentPointer
¼tcpHeaderBuffer[0]
¼tcpHeaderBuffer[1]
¼tcpHeaderBuffer[2]
¼tcpHeaderBuffer[3]
¼tcpHeaderBuffer[4]
¼(dataOffsetandReserve >>4) * 4 ¼tcpHeaderBuffer[5]
¼flags & 0x01
¼(flags>>1) & 0x01
¼(flags>>2) & 0x01
¼(flags>>3) & 0x01
¼(flags>>4) & 0x01
¼(flags>>5) & 0x01
¼(flags>>6) & 0x01
¼(flags>>7) & 0x01
¼tcpHeaderBuffer[6]
¼tcpHeaderBuffer[7]
¼tcpHeaderBuffer[8]

Now that we have the basics under our belt, let us look at:

How to place our NIC into Promiscuous Mode 1.
2. How to create a raw socket in Linux
3. How to unpack the packet to obtain the individual fields

We are ready to build an application that can capture network packets and extract the information that will allow us to monitor traffic.

PYTHON SILENT NETWORK MAPPING TOOL (PSNMT)

Now that we have the basics for sniffing a network packet, I need to parse the data and extract the information I need. For this example, I am not interested in collecting packets and simply printing the results, rather I want to achieve the following objectives:

(1) Collect IP addresses that are active on the network I am monitoring. (I plan to leave the monitor in place for a long period of time to capture network devices that only turn on periodically or sporadically.)

(2) Collect IP addresses of remote computers that are interacting with my local network. These could be web, mail, or a plethora of Cloud services.
(3) Collect service ports being used by local and/or remote computers. Specifically, I am interested in “Well Defined Ports”: 0-1023 or “Registered Ports”: 1024-49151.
(4) Next I wish to report only unique entries. In other words, if the local host 192.168.0.5 is discovered and is found to be using host port 80, I only want to see that unique entry once, not each time it is discovered.
(5) Finally, to limit the scope of the program, I want to collect only TCP or UDP packets within an IPv4 environment. The program can be easily expanded to handle other protocols and IPv6 in the future.

In order to meet the requirements stated above I only need to extract the following fields from the headers:

(1) Protocol
(2) Source IP address
(3) Destination IP address
(4) Source port
(5) Destination port

Examining Figures 9.4 and 9.5, the Protocol field, along with the Source and Destination IP addresses exist in the IPv4 header, while the Source and Destination ports are in the TCP header. This means I will have to parse out both headers to obtain the needed information. I have also included Figure 9.6 which depicts the UDP header, which I also use to handle UDP packet extraction.

There are several technical issues that need to be addressed along with the high level requirements:

(1) What type of data element should I use to store the information collected? a. I am going to use a simple list to hold the data collected from the packets and append data to the lists for each packet received.
ipObservations¼[]

UDP Header

Offsets Bytes 0 1 2 3
Octet BITS 0 1 2 3 4 5 6 7 8 9 1011 12131415161718 19202122232425 262728293031
00 **Source port Destination port**
4 32 Length Checksum

FIGURE 9.6
Typical UDP packet header.

(2) Since the socket.recvfrom()method is synchronous, how will I signal when to stop collection, and how will I limit the time of the collection activities? a. I am going to use the Python Standard Library signal module and integrate

this into the collection loop. I set this up by first creating a class myTimeout that will be raised by a handler when a specified time has expired. I then integrate the myTimeout exception handler into the try/except handler of the receive packet loop.

```
class myTimeout(Exception): pass

def handler(signum, frame):
print'timeout received', signum raise myTimeout()

# Set the signal handler
signal.signal(signal.SIGALRM, handler)

# set the signal to expire in n seconds signal.alarm(n)
...
...
try:

while True:
recvBuffer, addr¼mySocket.recvfrom(65535)

src,dst ¼decoder.PacketExtractor(recvBuffer,\ False)
sourceIPObservations.append(src)
destinationIPObservations.append(dst)
```

except myTimeout: pass

(3) How will I create only unique entries?
a. The code above will record every pair of source IP/Port and destination IP/Port, with a result being an unsorted list and will contain duplicate entries. To solve this problem, once the collection is complete, I will use a little knowledge of Python data types to help here. Once collection is completed (for the entire time frame), I first convert the list into a set, this will immediately collapse any duplicates (as this is a fundamental property of sets). Then I will convert the set back to a list and then sort the list.

uniqueSrc ¼set(map(tuple, ipObservations)) finalList¼ list(uniqueSrc)
finalList.sort()

(4) How should I output the results?
a. In order to provide a workable list, the program will generate a commaseparated value (CSV) file that can then be further processed or examined in a worksheet.

PSNMT SOURCE CODE

The source code is broken into the following five source files. Each of the files contains detailed comments describing all aspects of the program. Figure 9.7 depicts the WingIDE environment for this application.

Source Purpose

psnmt.py
decoder.py
_commandparser.py _csvHandler.py
_classLogging.py Main program setup and loop
Decoder for raw packets
Parser for the user command line Handler for creating/writing csv file output
Class for handling forensic logging

psnmt.py source code

#
Python Passive Network Monitor and Mapping Tool

Import Standard Library Modules import socket import signal import os import sys
network interface library used for raw sockets # generation of interrupt signals i.e. timeout # operating system functions i.e. file I/o # system level functions i.e. exit()

FIGURE 9.7
WingIDE environment for the PSNMT application.

# Import application specific Modules
import decoder
import _commandParser import _csvHandler # module to decode tcp and udp packets # parse out command line args
# output generation

from _classLogging import _ForensicLog # Logging operations
# Process the Command Line Arguments
userArgs¼_commandParser.ParseCommandLine()

# create a log object
logPath¼os.path.join(userArgs.outPath,"ForensicLog.txt")

oLog¼_ForensicLog(logPath)

oLog.writeLog("INFO", "PS-NMT Started")
csvPath¼os.path.join(userArgs.outPath,"ps-nmtResults.csv")
oCSV¼_csvHandler._CSVWriter(csvPath)
# Setup the protocol to capture
if userArgs.TCP:

PROTOCOL ¼socket.IPPROTO_TCP elif userArgs.UDP:
PROTOCOL¼socket.IPPROTO_UDP else:
print'Capture protocol not selected'
sys.exit()

# Setup whether output should be verbose
if userArgs.verbose:

VERBOSE ¼True else:
VERBOSE¼False

Calculate capture duration
captureDuration¼userArgs.minutes * 60
Create timeout class to handle capture duration
class myTimeout(Exception): pass
Create a signal handler that raises a timeout event # when the capture duration is reached

def handler(signum, frame):
print'timeout received', signum raise myTimeout()

# Enable Promiscious Mode on the NIC
ret¼os.system("ifconfig eth0 promisc")
if ret ¼¼ 0:

oLog .writeLog("INFO",'Promiscious Mode Enabled') # create an INET, raw socket
AF_INET specifies # SOCK_RAW specifies # IPPROTO_TCP or UDP ipv4 a raw protocol at the network layer Specifies the protocol to capture

try : mySocket¼socket.socket(socket.AF_INET, socket.SOCK_RAW, PROTOCOL)

```
oLog.writeLog("INFO",'Raw Socket Open')

except :
# if socket open fails
oLog.writeLog("ERROR",'Raw Socket Open Failed') del oLog
if VERBOSE:

print'Error Opening Raw Socket'
sys.exit()
# Set the signal handler to the duraton specified by the user
signal.signal(signal.SIGALRM, handler) signal.alarm(captureDuration)

# create a list to hold the results from the packet capture # I'm only interested in
Protocol Source IP, Source Port, Destination IP, Destination Port

ipObservations¼[]
# Begin receiving packets until duration is received # the inner while loop will
execute until the timeout
try:
while True:
# attempt recieve (this call will wait) recvBuffer, addr¼mySocket.recvfrom(255)
# decode the received packet
content¼decoder.PacketExtractor(recvBuffer, VERBOSE)
# append the results to our list ipObservations.append(content)

# write details to the forensic log file oLog.writeLog('INFO', \
'RECV:'þcontent[0]þ\
'SRC :'þcontent[1]þ\
'DST :'þcontent[3])

except myTimeout: pass

# Once time has expired disable Promiscous Mode ret¼os.system("ifconfig eth0
-promisc") oLog.writeLog("INFO",'Promiscious Mode Diabled')

# Close the Raw Socket
mySocket.close()
oLog.writeLog("INFO",'Raw Socket Closed')

# Create unique sorted list
```

uniqueSrc ¼set(map(tuple, ipObservations)) finalList¼list(uniqueSrc)
finalList.sort()
# Write each unique sorted packet to the csv file for packet in finalList:

oCSV.writeCSVRow(packet)
oLog.writeLog('INFO','Program End')

Close the Log and CSV objects del oLog
del oCSV

else:
print'Promiscious Mode not Set'
decoder.py source code
# Packet Extractor / Decoder Module #
import socket, sys from struct import *

Constants
PROTOCOL_TCP¼6 PROTOCOL_UDP¼17

PacketExtractor
#
# Purpose: Extracts fields from the IP, TCP and UDP Header #
# Input: packet: buffer from socket.recvfrom() method # displaySwitch: True: Display the details, False omits # Output: result list containing
# protocol, srcIP, srcPort, dstIP, dstPort #
def PacketExtractor(packet, displaySwitch):

#Strip off the first 20 characters for the ip header stripPacket¼packet[0:20]
#now unpack them
ipHeaderTuple¼unpack('!BBHHHBBH4s4s', stripPacket)
# unpack returns a tuple, for illustration I will extract # each individual values

verLen ¼ipHeaderTuple[0] dscpECN ¼ipHeaderTuple[1] # Field Contents
Field 0: Version and Length # Field 1: DSCP and ECN
packetLength ¼ipHeaderTuple[2] packetID
flagFrag
¼ipHeaderTuple[3] ¼ipHeaderTuple[4]

timeToLive protocol checkSum sourceIP destIP
¼ipHeaderTuple[5] ¼ipHeaderTuple[6] ¼ipHeaderTuple[7] ¼ipHeaderTuple[8]

¼ipHeaderTuple[9] # Field 2: Packet Length # Field 3: Identification # Field 4: Flags and Fragment Offset
Field 5: Time to Live (TTL) # Field 6: Protocol Number # Field 7: Header Checksum # Field 8: Source IP
Field 9: Destination IP

Calculate / Convert extracted values
version ¼verLen >>4 # Upper Nibble is the version

Number
length ¼verLen & 0x0F # Lower Nibble represents the
size
ipHdrLength ¼length * 4 # Calculate the header length in
bytes

# covert the source and destination address to typical dotted notation strings
sourceAddress¼socket.inet_ntoa(sourceIP);
destinationAddress¼socket.inet_ntoa(destIP);

if displaySwitch :
print'¼¼¼

using the unpack() function

tcpHeaderBuffer ¼unpack('!HHLLBBHHH', stripTCPHeader) sourcePort
destinationPort sequenceNumber acknowledgement
dataOffsetandReserve¼tcpHeaderBuffer[4]
tcpHeaderLength
tcpChecksum
¼(dataOffsetandReserve >>4)* 4 ¼tcpHeaderBuffer[7]
¼tcpHeaderBuffer[0] ¼tcpHeaderBuffer[1] ¼tcpHeaderBuffer[2]
¼tcpHeaderBuffer[3]

if displaySwitch : print
print'TCP Header' print'_ __ __ __ __ _ __ __ __ __ _ __ __'

print'Source Port: 'þstr(sourcePort) print'Destination Port :'þstr(destinationPort)
print'Sequence Number : 'þstr(sequenceNumber) print'Acknowledgement :
'þstr(acknowledgement) print'TCP Header Length: 'þstr(tcpHeaderLength)þ

'bytes '
print'TCP Checksum:'þhex(tcpChecksum)
print

return(['TCP', sourceAddress, sourcePort, destinationAddress, destinationPort])
elif protocol ¼¼ PROTOCOL_UDP:
stripUDPHeader¼packet[ipHdrLength:ipHdrLength + 8]
unpack returns a tuple, for illustration I will extract # each individual values
using the unpack() function
udpHeaderBuffer¼unpack('!HHHH', stripUDPHeader)

sourcePort ¼udpHeaderBuffer[0] destinationPort¼udpHeaderBuffer[1]
udpLength ¼udpHeaderBuffer[2] udpChecksum ¼udpHeaderBuffer[3]

if displaySwitch :
print
print'UDP Header'
print'_ __ __ __ _ __ __ __ __ _ __ __ __'

print'Source Port: 'þstr(sourcePort) print'Destination Port :'þstr(destinationPort)
print'UDP Length: 'þstr(udpLength)þ'bytes' print'UDP
Checksum:'þhex(udpChecksum) print

```
return(['UDP', sourceAddress, sourcePort, destinationAddress, destinationPort])
else:
# For expansion protocol support
if displaySwitch:
print'Found Protocol :'þstr(protocol)
return(['Unsupported',sourceAddress,0,\ destinationAddress,0])
```

commandParser.py

```
#
# PSNMT Argument Parser #

import argparse

import os # Python Standard Library - Parser for command-line options,
arguments
# Standard Library OS functions

# Name: ParseCommand() Function
#
# Desc: Process and Validate the command line arguments
# use Python Standard Library module argparse
#
# Input: none
#
# Actions:
# Uses the Standard Library argparse to process the command line #
def ParseCommandLine():

parser¼argparse.ArgumentParser('PS-NMT')
parser.add_argument('- v', '--verbose', help¼"Display packet details",
action¼'store_true')

# setup a group where the selection is mutually exclusive and required.
group¼parser.add_mutually_exclusive_group(required¼True)
group.add_argument('--TCP', help¼'TCP Packet Capture', action¼'store_true')
group.add_argument('--UDP', help¼'UDP Packet Capture', action¼'store_true')
parser.add_argument('- m','--minutes',help¼'Capture Duration in
minutes',type¼int)
```

parser.add_argument('- p', '--outPath', type¼ValidateDirectory, required¼True, help¼"Output Directory")

theArgs¼parser.parse_args()
return theArgs
End Parse Command Line ¼¼¼

```
def writeLog(self, logType, logMessage): if logType ¼¼ "INFO":
logging.info(logMessage)

elif logType ¼¼ "ERROR":
logging.error(logMessage)
elif logType ¼¼ "WARNING":
logging.warning(logMessage)
else:
logging.error(logMessage)
return

def __del__(self):
logging.info("Logging Shutdown") logging.shutdown()
```

csvHandler.py source code

```
import csv #Python Standard Library - for csv files

#
# Class: _CSVWriter
#
# Desc: Handles all methods related to comma separated value operations #
# Methods constructor: Initializes the CSV File
# writeCVSRow: Writes a single row to the csv file # writerClose: Closes the
CSV File

class _CSVWriter:

def __init__(self, fileName): try:
# create a writer object and then write the header row
self.csvFile¼open(fileName, 'w b')
self.writer¼csv.writer(self.csvFile, delimiter¼',', quoting¼csv.QUOTE_ALL)
self.writer.writerow(('Protocol','Source IP','Source Port', 'Destination
IP','Destination Port'))

except:
log.error('CSV File Failure')
def writeCSVRow(self, row):
self.writer.writerow((row[0], row[1], str(row[2]), row [3], str(row[4]) ) )
def __del__(self):
self.csvFile.close()
```

PROGRAM EXECUTION AND OUTPUT

As you can see, the psnmt Python application is constructed as a command line application. This makes sense for this type of application because it is likely to be executed as a cron job (scheduled to run at certain times for example).

The command line has the following parameters:
Parameter Purpose and Usage

-v Verbose: Writes intermediate results to standard output when specified
-m Minutes: Duration in minutes to perform collection activities
–TCP | –UDP Protocol: Defines the application protocol to capture
-p Produces: Defines the output directory for the forensic log and csv file

Example command line:
sudo Python psnmt –v –TCP –m 60 –p /home/chet/Desktop

Note: sudo (is used to force the execution of the command with administrative rights, which is required). This command will capture TCP packets for 60 min, produce verbose output and create a log and csv file on the user's desktop.

Figures 9.8 and 9.9 show sample executions for both TCP and UDP captures. These runs created both csv and forensic log files and entries.

```
chet@PythonForensics: ~/Desktop/Chapter 9
chet@PythonForensics:~/Desktop/Chapter 9$ sudo python ps-nmt.py -v -m 2 --TCP -p /home/chet/Desktop/Chapter\ 9
IP HEADER
-----------------------
Version:          4
Packet Length:    60 bytes
Header Length:    20 bytes
TTL:              43
Protocol:         6
Checksum:         0x5227
Source IP:        74.125.196.147
Destination IP:   192.168.0.25

TCP Header
-------------------
Source Port:        443
Destination Port :  56652
Sequence Number :   621577023
Acknowledgement :   2310411757
TCP Header Length:  40 bytes
TCP Checksum:       0x6d8c

IP HEADER
-----------------------
Version:          4
Packet Length:    60 bytes
Header Length:    20 bytes
TTL:              43
Protocol:         6
Checksum:         0xf9f3
Source IP:        74.125.196.147
Destination IP:   192.168.0.25

TCP Header
-------------------
Source Port:        443
Destination Port :  56653
Sequence Number :   1824497575
Acknowledgement :   1049873182
TCP Header Length:  40 bytes
TCP Checksum:       0x6d67
```

FIGURE 9.8
psnmt TCP sample run.

```
chet@PythonForensics: ~/Desktop/Chapter 9

chet@PythonForensics:~/Desktop/Chapter 9$ sudo python psnmt.py -v -m 2 --UDP -p '/home/chet/Desktop/Chapter 9'
IP HEADER
-----------------------
Version:          4
Packet Length:    61 bytes
Header Length:    20 bytes
TTL:              64
Protocol:         17
Checksum:         0x98fe
Source IP:        127.0.0.1
Destination IP:   127.0.0.1

UDP Header
-------------------
Source Port:        52309
Destination Port :  53
UDP Length:         41 bytes
UDP Checksum:       0xfe3c

IP HEADER
-----------------------
Version:          4
Packet Length:    121 bytes
Header Length:    20 bytes
TTL:              58
Protocol:         17
Checksum:         0x86f7
Source IP:        66.153.128.98
Destination IP:   192.168.0.25

UDP Header
-------------------
Source Port:        53
Destination Port :  23992
UDP Length:         101 bytes
UDP Checksum:       0x474f

IP HEADER
```

FIGURE 9.9
psnmt UDP sample run.

Forensic log
TCP capture example

2014-01-19 11:29:51,050 PS-NMT Started
2014-01-19 11:29:51,057 Promiscious Mode Enabled
2014-01-19 11:29:51,057 Raw Socket Open
2014-01-19 11:29:55,525 RECV: TCP SRC : 173.194.45.79 DST : 192.168.0.25
2014-01-19 11:29:55,526 RECV: TCP SRC : 173.194.45.79 DST : 192.168.0.25
2014-01-19 11:29:56,236 RECV: TCP SRC : 74.125.196.147 DST : 192.168.0.25
2014-01-19 11:29:56,270 RECV: TCP SRC : 74.125.196.147 DST : 192.168.0.25
2014-01-19 11:29:56,270 RECV: TCP SRC : 74.125.196.147 DST : 192.168.0.25

2014-01-19 11:29:56,271 RECV: TCP SRC : 74.125.196.147 DST : 192.168.0.25
2014-01-19 11:29:56,527 RECV: TCP SRC : 74.125.196.147 DST : 192.168.0.25
2014-01-19 11:29:56,543 RECV: TCP SRC : 74.125.196.147 DST : 192.168.0.25
2014-01-19 11:29:56,544 RECV: TCP SRC : 74.125.196.147 DST : 192.168.0.25
2014-01-19 11:29:56,546 RECV: TCP SRC : 74.125.196.147 DST : 192.168.0.25
2014-01-19 11:30:37,437 RECV: TCP SRC : 66.153.250.240 DST : 192.168.0.25
2014-01-19 11:30:37,449 RECV: TCP SRC : 66.153.250.240 DST : 192.168.0.25
2014-01-19 11:30:54,546 RECV: TCP SRC : 173.194.45.79 DST : 192.168.0.25
2014-01-19 11:30:55,454 RECV: TCP SRC : 74.125.196.147 DST : 192.168.0.25
2014-01-19 11:31:35,487 RECV: TCP SRC : 66.153.250.240 DST : 192.168.0.25
2014-01-19 11:31:51,063 Promiscious Mode Diabled
2014-01-19 11:31:51,064 Raw Socket Closed 2014-01-19 11:31:51,064 Program End 2014-01-19 11:31:51,064 Logging Shutdown

UDP capture example

2014-01-19 13:27:09,366 PS-NMT Started
2014-01-19 13:27:09,371 Promiscious Mode Enabled
2014-01-19 13:27:09,372 Raw Socket Open
2014-01-19 13:27:09,528 Logging Shutdown
2014-01-19 13:36:33,472 PS-NMT Started
2014-01-19 13:36:33,477 Promiscious Mode Enabled
2014-01-19 13:36:33,477 Raw Socket Open
2014-01-19 13:36:45,234 Logging Shutdown
2014-01-19 13:37:51,748 PS-NMT Started
2014-01-19 13:37:51,754 Promiscious Mode Enabled
2014-01-19 13:37:51,754 Raw Socket Open
2014-01-19 13:37:59,534 RECV: UDP SRC : 127.0.0.1 DST : 127.0.0.1
2014-01-19 13:37:59,546 RECV: UDP SRC : 66.153.128.98 DST : 192.168.0.25
2014-01-19 13:37:59,546 RECV: UDP SRC : 127.0.0.1 DST : 127.0.0.1

2014-01-19 13:37:59,549 RECV: UDP SRC : 66.153.162.98 DST : 192.168.0.25
2014-01-19 13:38:09,724 RECV: UDP SRC : 127.0.0.1 DST : 127.0.0.1
2014-01-19 13:38:09,879 RECV: UDP SRC : 66.153.128.98 DST : 192.168.0.25
2014-01-19 13:38:09,880 RECV: UDP SRC : 127.0.0.1 DST : 127.0.0.1
2014-01-19 13:38:10,387 RECV: UDP SRC : 127.0.0.1 DST : 127.0.0.1
2014-01-19 13:38:10,551 RECV: UDP SRC : 66.153.128.98 DST : 192.168.0.25
2014-01-19 13:38:10,551 RECV: UDP SRC : 127.0.0.1 DST : 127.0.0.1
2014-01-19 13:38:45,114 RECV: UDP SRC : 66.153.128.98 DST : 192.168.0.25
2014-01-19 13:38:46,112 RECV: UDP SRC : 66.153.128.98 DST : 192.168.0.25
2014-01-19 13:39:45,410 RECV: UDP SRC : 66.153.128.98 DST : 192.168.0.25
2014-01-19 13:39:51,760 Promiscious Mode Diabled
2014-01-19 13:39:51,761 Raw Socket Closed
2014-01-19 13:39:51,761 Program End
2014-01-19 13:39:51,761 Logging Shutdown

CSV file output example

Figures 9.10 and 9.11 depict a sample output of the CSV file created by PSNMT.

	A	B	C	D	E
1	**Protocol**	**Source IP**	**Source Port**	**Destination IP**	**Destination Port**
2	TCP	127.0.0.1	36480	127.0.0.1	54792
3	TCP	127.0.0.1	54792	127.0.0.1	36480
4	TCP	66.153.25(	443	192.168.0.25	35027
5	TCP	74.125.19(	443	192.168.0.25	56580
6	TCP	74.125.19(	443	192.168.0.25	56581

FIGURE 9.10
Sample TCP output file shown in Excel.

	A	B	C	D	E
1	Protocol	Source IP	Source Port	Destination IP	Destination Port
2	UDP	127.0.0.1	53	127.0.0.1	35633
3	UDP	127.0.0.1	53	127.0.0.1	51420
4	UDP	127.0.0.1	53	127.0.0.1	52309
5	UDP	127.0.0.1	35633	127.0.0.1	53
6	UDP	127.0.0.1	51420	127.0.0.1	53
7	UDP	127.0.0.1	52309	127.0.0.1	53
8	UDP	66.153.128	53	192.168.0.25	11303
9	UDP	66.153.128	53	192.168.0.25	23992
10	UDP	66.153.128	53	192.168.0.25	35021
11	UDP	66.153.128	53	192.168.0.25	43421
12	UDP	66.153.128	53	192.168.0.25	56857
13	UDP	66.153.128	53	192.168.0.25	58487
14	UDP	66.153.162	53	192.168.0.25	23992

FIGURE 9.11
Sample UDP output file shown in Excel.

CHAPTER REVIEW

In this chapter, I introduced the raw sockets and how they can be utilized to capture network packets. In order to do so, I explained the Promiscuous Mode of modern network adapters and demonstrated how to configure a NIC for this operation. I also discussed the importance of network sniffing in order to more thoroughly map and monitor network activities. I used the unpack() function to extract each field from the IPv4, TCP and UDP packets and described in detail, how to apply this to buffered data. Finally, I created an application that collects and records either TCP or UDP traffic isolating unique Source IP, Source Port, Destination IP, and Destination Port combinations. This Python application can be controlled through command line arguments in order to easily integrate with cron jobs or other scheduling mechanisms. As part of the command line, I used a signaling mechanism to stop packet collection after a specified time period has expired.

SUMMARY QUESTION/CHALLENGE

I decided to pose only one question in this chapter, more of a challenge problem actually that will take the PSNMT application to the next level:

In order to make this current application more useful, the translation of port 1. numbers into well-known services is necessary in order to determine the likely use or uses of each IP address that we extracted from the captured packets. Using the reserved and well-known port usage maps provided and defined by the Internet Engineering Task Force [IETF], (or by reading /etc/services) expand the

PSNMT application to isolate each local IP address and list the services that they are likely running on each. When possible, attempt to broadly classify the operating system behind each IP address through the examination of port usage (at least classify each IP as Windows, Linux, or other).

Additional Resources
http://docs.Python.org/2/library/struct.html?highlight¼unpack#struct.unpack.
http://www.ietf.org/assignments/service-names-port-numbers/service-names-port-numbers. txt.
CHAPTER

Multiprocessing for Forensics 10

CHAPTER CONTENTS

Introduction 265
What is Multiprocessing? 266
Python Multiprocessing Support 266
Simplest Multiprocessing Example 269
Single Core File Search Solution 270
Multiprocessing File Search Solution 270
Multiprocessing File Hash 272
Single Core Solution 272
Multi-core Solution A 274
Multi-core Solution B 277
Multiprocessing Hash Table Generation

..... 279
Single Core Password Generator Code 280
Multi-core Password Generator 283
Multi-core Password Generator Code 283
Chapter Review .....
288
Summary Question/Challenge
..... 288
Additional Resources .....
288

INTRODUCTION

As the footprint of digital evidence and digital crime widens, our ability to perform examinations in a timely fashion is clearly strained. According to DFI News “The backlog of caseloads from law enforcement agencies has grown from weeks to months worldwide. Digital forensic specialists cannot be trained fast enough and the number of specialists required to analyze the mountains of digital evidence in common crimes is simply beyond budget constraints” (DFI, 2013). Many digital investigation tools today are single threaded, meaning that the software can only execute one command at a time. These tools were invariably originally developed before multi-core processors were commonplace. In this chapter, I introduce the multiprocessing capabilities of Python that relate to some common forensic challenges. In Chapter 11, I will transfer these applications to the cloud and demonstrate how to increase performance by expanding access to additional cores and allowing forensic operations to be performed on a cloud platform.

© 2014 Elsevier Inc. All rights reserved.

WHAT IS MULTIPROCESSING?

Simply stated multiprocessing is the simultaneous execution of a program on two or more central processing units (CPUs) or cores. In order to provide significant performance improvements, the developer of forensic applications must define areas of the code that have the following characteristics:

The code is processor-intensive.(1)

(2) It is possible to break the code apart into separate processing threads that can

execute in parallel.
(3) The processing between threads can be load-balanced. In other words, the goal is to distribute processing such that each of the threads complete at approximately the same time.

As you may already surmise, the problem with general purpose forensic tools is this: if they were not engineered to meet the objectives above from the start, adapting them to modern multi-core architectures may prove to be difficult. In addition, licensing of such technologies to run simultaneously on multi-cores or in the cloud may be cost-prohibitive. Finally, many of the most commonly used forensic tools operate within the Windows environment which is not ideally suited to run simultaneously on thousands of cores.

PYTHON MULTIPROCESSING SUPPORT

The Python Standard Library includes the package “multiprocessing” (Python multiprocessing module). Using the Python Standard Library for multiprocessing is a great place to begin multiprocessing and will ensure compatibility across a wide range of computing platforms, including the cloud. We utilize the multiprocessing package in typical fashion starting by importing the package multiprocessing. After import of the package, I execute the help(multiprocessing) function to reveal the details. I have eliminated some of the extraneous data produced by help, and I have highlighted the functions and classes that I will be utilizing to develop a couple of multiprocessing examples.

```
import multiprocessing
help(multiprocessing)
# NOTE this is only a partial excerpt from the output # of the Help command
Help on package multiprocessing:
NAME multiprocessing
```



```
PACKAGE CONTENTS connection
dummy (package) forking
heap
managers
pool
process
queues
```

reduction
sharedctypes synchronize util

CLASSES
class Process(__builtin__.object)

| Process objects represent activity that is run in a separate process
|
| The class is analagous to'threading.Thread'
|
| Methods defined here:
|
| __init__(self, group¼None, target¼None, name¼None, args¼(), kwargs¼{})
|
| __repr__(self)
|
| is_alive(self)
| Return whether process is alive
|
| join(self, timeout¼None)
| Wait until child process terminates
|
| run(self)
| Method to be run in sub-process; can be overridden in sub-class
|
| start(self)
| Start child process
|
| terminate(self)
| Terminate process; sends SIGTERM signal or uses TerminateProcess()
|
FUNCTIONS
Array(typecode_or_type, size_or_initializer, **kwds) Returns a synchronized shared array
BoundedSemaphore(value¼1)
Returns a bounded semaphore object

Condition(lock¼None)
Returns a condition object

Event()
Returns an event object
JoinableQueue(maxsize¼0) Returns a queue object
Lock()
Returns a non-recursive lock object
Manager()
Returns a manager associated with a running server process
The managers methods such as 'Lock()', 'Condition()' and 'Queue()' can be used to create shared objects.
Pipe(duplex¼True)
Returns two connection object connected by a pipe
Pool(processes¼None, initializer¼None, initargs¼(), maxtasksperchild¼None)
Returns a process pool object
Queue(maxsize¼0)
Returns a queue object
RLock()
Returns a recursive lock object
RawArray(typecode_or_type, size_or_initializer) Returns a shared array
RawValue(typecode_or_type, *args) Returns a shared object
Semaphore(value¼1)
Returns a semaphore object
Value(typecode_or_type, *args, **kwds) Returns a synchronized shared object

active_children()
Return list of process objects corresponding to live child processes

allow_connection_pickling()
Install support for sending connections and sockets between processes

cpu_count()
Returns the number of CPUs in the system
current_process()
Return process object representing the current process

freeze_support()
Check whether this is a fake forked process in a frozen executable. If so then run code specified by commandline and exit.

VERSION

0.70a1
AUTHOR
R. Oudkerk (r.m.oudkerk@gmail.com)

One of the first functions within the multiprocessing package you will be interested in is cpu_count(). Before you can distribute processing among multiple cores, you need to know how many cores you have access to.

```
import multiprocessing
multiprocessing.cpu_count()
4
```

As you can see on my Windows laptop, I have four CPU cores to work with; it would be a shame if my Python code only ran on one of those cores. Without applying the multiprocessing package and designing multi-core processing methods into my algorithm, my Python code would run on exactly one core.

Let us move on to our first example of applying multiprocessing within a Python program that has some forensic implication and relevance.
SIMPLEST MULTIPROCESSING EXAMPLE

In this first example I chose to develop the simplest multiprocessing example I could think of that would maximize the use of the four cores I have on my laptop. The program has a single function named SearchFile(), the function takes two parameters: (1) a filename and (2) the string that I wish to search for in that file. I am using a simple text file as my search target that contains a dictionary of words. The file is 170 MB in size which even on modern systems should create some I/O latency. I provide two examples, first a program that does not employ multiprocessing and simply makes four successive calls to the SearchFile function. The second example creates four processes and distributes the processing evenly between the four cores.

Single core file search solution

```
# Simple Files Search Single Core Processing
import time

def SearchFile(theFile, theString): try:
fp ¼ open(theFile,'r')
buffer ¼ fp.read()
```

```
fp.close()
if theString in buffer:

print'File:', theFile,'String:',\ theString,'\t','Found' else:
print'File:', theFile,'String:', \ theString,'\t','Not Found' except:
print'File processing error'
startTime ¼ time.time()

SearchFile('c:\\TESTDIR\\Dictionary.txt','thought')
SearchFile('c:\\TESTDIR\\Dictionary.txt','exile')
SearchFile('c:\\TESTDIR\\Dictionary.txt','xavier')
SearchFile('c:\\TESTDIR\\Dictionary.txt','$Slllb!')

elapsedTime ¼ time.time() - startTime print'Duration:', elapsedTime
# Program Output

File: c:\TESTDIR\Dictionary.txt String: thought Found File:
c:\TESTDIR\Dictionary.txt String: exile Found File: c:\TESTDIR\Dictionary.txt
String: xavier Found File: c:\TESTDIR\Dictionary.txt String: $Slllb! Not Found
Duration: 4.3140001297 Seconds
```

Multiprocessing file search solution

The multiprocessing solution is depicted below and as you can see the performance is substantially better even considering the file I/O aspects of opening, reading, and closing the associated file each time.

```
# Simple Files Search MultiProcessing
from multiprocessing import Process import time

def SearchFile(theFile, theString): try:
fp ¼ open(theFile,'r')
buffer¼ fp.read()
fp.close()
if theString in buffer:

print'File:', theFile,'String:', theString,

'\t','Found'
else:
```

```
print 'File: ', theFile, ' String: ', theString,
'\t', ' Not Found'
except:
print'File processing error'

#
# Create Main Function #

if __name__¼¼'__main__':
startTime ¼ time.time()

p1 ¼ Process(target¼SearchFile,
\args¼('c:\\TESTDIR\\Dictionary.txt','thought') )
p1.start()

p2 ¼ Process(target¼SearchFile, \
args¼('c:\\TESTDIR\\Dictionary.txt','exile') )
p2.start()

p3 ¼ Process(target¼SearchFile, \
args¼('c:\\TESTDIR\\Dictionary.txt','xavier') )
p3.start()

p4 ¼ Process(target¼SearchFile, \
args¼('c:\\TESTDIR\\Dictionary.txt','$Slllb') )
p4.start()

# Next we use the join to wait for all processes to complete

p1.join() p2.join() p3.join() p4.join()

elapsedTime ¼ time.time() - startTime print'Duration:', elapsedTime
# Program Output

File: c:\TESTDIR\Dictionary.txt String: thought Found File:
c:\TESTDIR\Dictionary.txt String: exile Found File: c:\TESTDIR\Dictionary.txt
String: xavier Found File: c:\TESTDIR\Dictionary.txt String: $Slllb Not Found
Duration: 1.80399990082
```

MULTIPROCESSING FILE HASH

Certainly one of the most frequently used forensic tools is the one-way cryptographic hash. As you already know Python includes a hashing library as part of the Standard Library. As an experiment, I am going to perform a SHA512 Hash of four separate instances of the same file using a nonmultiprocessing method. I will set up a timer as well to calculate the elapsed time to perform the single-threaded approach.

Single core solution

```
# Single Threaded File Hasher

import hashlib import os
import sys
import time

# Create a constant for the local directory
HASHDIR ¼ 'c:\\HASHTEST\\'
# Create an empty list to hold the resulting hash results results ¼ []

try: # Obtain the list of files in the HASHDIR listOfFiles ¼ os.listdir(HASHDIR)

# Mark the starting time of the main loop startTime ¼ time.time()
for eachFile in listOfFiles:

# Attempt File Open
fp ¼ open(HASHDIR+eachFile,'rb') # Then Read the contents into a buffer
fileContents ¼ fp.read()

# Close the File fp.close()
# Create a hasher object of type sha256 hasher¼ hashlib.sha256()
# Hash the contents of the buffer hasher.update(fileContents)
# Store the results in the results list
results.append([eachFile, hasher.hexdigest()])
# delete the hasher object del hasher
# Once all the files have been hashed calculate # the elapsed time
elapsedTime ¼ time.time() - startTime
except:
# If any exceptions occur notify the user and exit
```

```
print('File Processing Error') sys.exit(0)
# Print out the results
# Elapsed Time in Seconds and the Filename / Hash Results
print('Elapsed Time:', elapsedTime) for eachItem in results:
print eachItem

# Program Output
# Note: Each File Processed is identical # With a Size: 249 MB

Elapsed Time: 27.8510000705719 Seconds
['image0.raw',
'41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b']
['image1.raw',
'41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b']
['image2.raw',
'41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b']
['image3.raw',
'41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b']
```

Multi-core solution A

Much of the art associated with multiprocessing is decomposing a problem into slices that can run simultaneously across multiple cores. Once each of the individual slices have completed, the results from each slice are merged into a final result. The biggest mistake that is typically made is the lack of careful consideration of these two points. Purchasing systems with multi-cores and then just running any solution on that multi

core system does not necessarily buy you much over a single core solution.

Using our single solution as a baseline, I will create a multi-core solution that leverages the four cores available on my laptop. For this first example, I will instantiate the object Process, one for each core. When creating each Process object, I only need to provide two parameters: (1) the target¼which in this case is the name of the function that will be called when the object.start() method is called. (2) the args¼which is the argument tuple to be passed to the function. In this example, only one argument is passed, namely, the name of the file to hash. Once I have instantiated an object, I execute the object.start() method to kick off each process. Finally, I utilize the object.join() method for each object; this causes execution to be halted in the main() process until the processes have

completed. You can include a parameter with object.join() which is a timeout value for the process. For example object.join(20) would allow the process 20 s to complete.

```
# Multiprocessing File Hasher A

import hashlib
import os
import sys
import time
import multiprocessing

# Create a constant for the local directory HASHDIR ¼'c:\\HASHTEST\\'

#
# hashFile Function, designed for multiprocessing #
# Input: Full Pathname of the file to hash #
#

def hashFile(fileName):
try:
fp ¼ open(fileName,'rb')

# Then Read the contents into a buffer fileContents ¼ fp.read()
# Close the File fp.close()

# Create a hasher object of type sha256 hasher¼ hashlib.sha256()
# Hash the contents of the buffer hasher.update(fileContents)
print(fileName, hasher.hexdigest())
# delete the hasher object del hasher

except:
# If any exceptions occur notify the user and exit print('File Processing Error')
sys.exit(0)

return True

#
# Create Main Function #
```

if __name__¼¼'__main__':
# Obtain the list of files in the HASHDIR listOfFiles ¼ os.listdir(HASHDIR)
# Mark the starting time of the main loop startTime ¼ time.time()
# create 4 sub-processes to do the work # one of each core in this test

Each Process contains:
Target function hashFile() it this example # Filename: picked from the list generated # by os.listdir()
once again an instance of the # 249 MB file is used
#
Next we start each of the processes

coreOne ¼ multiprocessing.Process(target¼hashFile, args¼(HASHDIR+listOfFiles[0],)) coreOne.start()

coreTwo ¼ multiprocessing.Process(target¼hashFile, args¼(HASHDIR+listOfFiles[1],))
coreTwo.start()

coreThree ¼ multiprocessing.Process(target¼hashFile, args¼(HASHDIR+listOfFiles[2],))
coreThree.start()

coreFour ¼ multiprocessing.Process(target¼hashFile, args¼(HASHDIR+listOfFiles[3],))
coreFour.start()

In this example, I was leveraging the knowledge of the hardware I was running the application on in order to maximize the distribution of processing. The Python multiprocessing library will automatically handle the distribution of the processes as cores become available. Also, since I could determine the number of cores available by using (multiprocessing.cpu_count()). I could have used that information to manually distribute the processing as well.

Next we use join to wait for all processes to complete

coreOne.join() coreTwo.join() coreThree.join() coreFour.join()

Once all the processes have completed and files have been # hashed and results

```
printed
# I calculate the elapsed time

elapsedTime ¼ time.time() - startTime
print('Elapsed Time:', elapsedTime)

# Program Output
# Note: Each File Processed is identical # With a Size: 249 MB

c:\\HASHTEST\\image2.raw
41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b
c:\\HASHTEST\\image1.raw
41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b
c:\\HASHTEST\\image3.raw
41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b
c:\\HASHTEST\\image0.raw
41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b
Elapsed Time: 8.40999984741211 Seconds
```

As you can see by distributing the processing across the four cores produced the desired results by significantly improving the performance of the hashing operations.

Multi-core solution B

Here is another option for distributing processing among multiple cores that yields a small performance improvement over the multi-core solution A, and additionally provides a much cleaner implementation when you are calling the same functions with different parameters. This is a likely scenario if you consider that in your design for forensic applications. Additionally, this solution only requires the use of a single class: pool that can handle the entire multi-core processing operations in a single line of code. I have highlighted the simplified implementation below.

```
# Multiprocessing File Hasher B

import hashlib
import os
import sys
import time
```

```
import multiprocessing

# Create a constant for the local directory HASHDIR ¼'c:\\HASHTEST\\'

#
# hashFile Function, designed for multiprocessing #
# Input: Full Pathname of the file to hash #
#

def hashFile(fileName):
try:
fp ¼ open(fileName,'rb')
# Then Read the contents into a buffer fileContents ¼ fp.read()
# Close the File fp.close()
# Create a hasher object of type sha256 hasher¼ hashlib.sha256()
# Hash the contents of the buffer hasher.update(fileContents) print(fileName,
hasher.hexdigest()) # delete the hasher object del hasher

except:
# If any exceptions occur notify the user and exit print('File Processing Error')
sys.exit(0)

return True

#
# Create Main Function #

if __name__ ¼¼'__main__':
# Obtain the list of files in the HASHDIR listOfFiles ¼ os.listdir(HASHDIR)
# Mark the starting time of the main loop startTime ¼ time.time()
# Create a process Pool with 4 processes mapping to # the 4 cores on my laptop
corePool ¼ multiprocessing.Pool(processes¼4)

# Map the corePool to the hashFile function
results ¼ corePool.map(hashFile, (HASHDIR+listOfFiles[0],\
HASHDIR+listOfFiles[1],\
HASHDIR+listOfFiles[2],\
HASHDIR+listOfFiles[3],))
```

Once all the files have been hashed and results printed # I calculate the elapsed time
elapsedTime ¼ time.time() - startTime
print('Elapsed Time:', elapsedTime,'Seconds')

Program Output
Note: Each File Processed is identical # With a Size: 249 MB

Elapsed Time: , 8.138000085830688, Seconds c:\\HASHTEST\\image0.raw, 41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b c:\\HASHTEST\\image2.raw, 41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b c:\\HASHTEST\\image1.raw, 41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b c:\\HASHTEST\\image3.raw, 41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b

Reviewing the results of the three implementations we find the following:
Processing
Implementation time MB (s) Notes
Single core solution 27.851 35.76
Multi-core solution A 8.409 118.44

Multi-core solution B 8.138 122.39 Typical implementation today processing each file in sequence Use of Process Class along with start and join methods Use of Pool Class simplifying implementation for common function processing yielding slightly better performance

MULTIPROCESSING HASH TABLE GENERATION

Rainbow Tables have been around for quite some time, providing a way to convert known hash values into possible password equivalents. In other words, they provide a way to lookup a hash value and associate that hash with a string that would generate that hash. Since hash algorithms are one-way, for all practical purposes two strings of characters will not generate the same hash value. They are highly collision-resistant. Rainbow Tables are utilized to crack passwords to operating systems, protected documents, and network user logins.

In recent years, the process of salting hashes has made Rainbow Tables less

effective as new tables would have to be generated as salt values are modified. These tables work because each password is hashed using the same process. In other words, two identical passwords would have the exact same hash value.

Today, this is prevented by randomizing hashed passwords, such that if two users use the same password, the hash values would be different. This is typically accomplished by appending a random string called a salt to a password prior to hashing.

Password ¼“letMeIn”
Salt¼‘&*6542Jk’
Combined Data to be Hashed: “&*6542JkletMeIn”
The salt does not need to be kept confidential as the purpose is to thwart the application of a rainbow or lookup table.

This means we need a fast way to generate more dynamic lookup tables. This is where multiprocessing can assist. Python actually has unique language mechanisms that support the generation of permutations and combinations built directly into the language that assists the developer. I will be using the itertools Standard Library (Python intertools module) to create the set of brute force passwords that I am looking for. The itertools module implements a number of iteration-like building blocks using a high-performance yet memory-optimized toolkit. Instead of having to develop permutation, combinations or product-based algebras, the library provides this for you.

First, I will create a simple Rainbow Table generator using a single core solution.
Single core password generator code
Single Core Password Table Generator
import standard libraries

import hashlib # Hashing the results
import time # Timing the operation import itertools # Creating controled combinations

#
Create a list of lower case, upper case, numbers # and special characters to include in the password table

lowerCase ¼ ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'] upperCase ¼ ['G', 'H', 'I', 'J', 'K', 'L']
numbers ¼ ['0', '1', '2', '3']

special ¼ ['!', '@', '#', '$']

combine to create a final list
allCharacters ¼ []
allCharacters ¼ lowerCase + upperCase + numbers + special

Define Directory Path for the password file
DIR ¼'C:\\PW\\'
Define a hypothetical SALT value SALT ¼ '&45Bvx9"

Define the allowable range of password length PW_LOW ¼ 2
PW_HIGH ¼ 6

Mark the start time startTime ¼ time.time() # Create an empty list to hold the final passwords pwList¼ []

create a loop to include all passwords # within the allowable range

for r in range(PW_LOW, PW_HIGH):
#Apply the standard library interator
#The product interator will generate the cartesian product # for allCharacters repeating for the range of
# PW_LOW to PW_HIGH
for s in itertools.product(allCharacters, repeat¼r):

# append each generated password to the
# final list
pwList.append(".join(s))

For each password in the list generate # generate a file containing the # hash, password pairs
one per line

try: # Open the output file fp ¼ open(DIR+'all','w')

process each generated password

for pw in pwList:
# Perform hashing of the password md5Hash ¼ hashlib.md5()

```
md5Hash.update(SALT+pw)
md5Digest ¼ md5Hash.hexdigest() # Write the hash, password pair to the file
fp.write(md5Digest +"+ pw +'\n') del md5Hash

except:
print'File Processing Error'
fp.close()

# Now create a dictionary to hold the # Hash, password pairs for easy lookup
pwDict¼ {}

try: # Open each of the output file fp ¼ open(DIR+'all','r')
# Process each line in the file which # contains key, value pairs
for line in fp:

# extract the key value pairs # and update the dictionary
pairs ¼ line.split()
pwDict.update({pairs[0] : pairs[1]})

fp.close()
except:
print'File Handling Error'
fp.close()

# When complete calculate the elapsed time

elapsedTime ¼ time.time() - startTime print'Elapsed Time:', elapsedTime
print'Passwords Generated:', len(pwDict) print

# print out a few of the dictionary entries # as an example

cnt ¼ 0
for key,value in (pwDict.items()): print key, value
cnt +¼ 1
if cnt > 10:
break;

print

# Demonstrate the use of the Dictionary to Lookup a password using a known
```

hash
Lookup a Hash Value

pw ¼ pwDict.get('c6f1d6b1d33bcc787c2385c19c29c208') print'Hash Value Tested¼ c6f1d6b1d33bcc787c2385c19c29c208' print'Associated Password¼ '+ p w

Program Output
Elapsed Time: 89.117000103 Passwords Generated: 5399020

3e47ac3f51daffdbe46ab671c76b44fb K23IH
a5a3614f49da18486c900bd04675d7bc $@fL1
372da5744b1ab1f99376f8d726cd2b7c hGfdd
aa0865a47331df5de01296bbaaf4996a 21ILG
c6f1d6b1d33bcc787c2385c19c29c208 #I#$$
c3c4246114ee80e9c454603645c9a416 #bg
6ca0e4d8f183c6c8b0a032b09861c95a L1H21
fd86ec2191f415cdb6c305da5e59eb7a HJg@h
335ef773e663807eb21d100e06b8c53e a$HbH
d1bae7cd5ae09903886d413884e22628 ba21H
a2a53248ed641bbd22af9bf36b422321 GHcda

Hash Value Tested¼ 2bca9b23eb8419728fdeca3345b344fc Associated Password¼ #I#$$

As you can see the code is quite succinct and straightforward, it leverages the itertools and hashing library to generate a brute force list. The result is a reference-able list of over 5.3 million salted passwords in under 90 seconds.

Multi-core password generator
Now that I have created a successful model for generating password combinations and created a dictionary with the resulting key/value pairs, I want to once again apply multiprocessing in order to generate a scalable solution. As you can see from the single core solution, it required just under 90 seconds to generate a little more than 5.3 million pairs. As we expand the number of characters to include in our generation and expand the size of allowable passwords beyond 5, the number of combinations grows exponentially. As I mentioned earlier, setting up an approach that lends itself to multiprocessing is the key. Now let us examine how well this actually scales by reviewing the

multi-core solution.

Multi-core password generator code

```
# Multi-Core Password Table Generator
# import standard libraries

import hashlib
import time
import itertools
import multiprocessing # Hashing the results
# Timing the operation
# Creating controled combinations # Multiprocessing Library

# # Create a list of lower case, upper case, numbers # and special characters to
include in the password table #

lowerCase ¼ ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'] upperCase ¼ ['G', 'H', 'I', 'J', 'K', 'L']
numbers ¼ ['0', '1', '2', '3']
special ¼ ['!', '@', '#', '$']
# combine to create a final list
allCharacters ¼ []
allCharacters ¼ lowerCase + upperCase + numbers + special

# Define Directory Path for the password files DIR ¼'C:\\PW\\'
# Define a hypothetical SALT value SALT ¼ '&45Bvx9"
# Define the allowable range of password length
PW_LOW¼ 2 PW_HIGH ¼ 6
def pwGenerator(size):
pwList¼ []
# create a loop to include all passwords # with a length of 3-5 characters

for r in range(size, size+1):
#Apply the standard library interator
for s in itertools.product(allCharacters, repeat¼r):

# append each generated password to the
# final list
pwList.append(".join(s))
```

For each password in the list generate # an associated md5 hash and utilize the
hash as the key

try: # Open the output file
fp ¼ open(DIR+str(size),'w')

process each generated password

for pw in pwList:
# Perform hashing of the password md5Hash ¼ hashlib.md5()
md5Hash.update(SALT+pw)
md5Digest ¼ md5Hash.hexdigest()
Write the hash, password pair to the file fp.write(md5Digest +"+ pw +'\n') del
md5Hash

except:
print'File Processing Error'
finally:
fp.close()

#
Create Main Function

if __name__¼¼'__main__':
# Mark the starting time of the main loop startTime ¼ time.time()
#create a process Pool with 4 processes corePool ¼
multiprocessing.Pool(processes¼4)
#map corePool to the Pool processes
results ¼ corePool.map(pwGenerator, (2, 3, 4, 5))
Create a dictionary for easy lookups pwDict¼ {}
For each file

for i in range(PW_LOW, PW_HIGH): try:
# Open each of the output files
fp ¼ open(DIR+str(i),'r')
# Process each line in the file which # contains key, value pairs
for line in fp:

extract the key value pairs

```
# and update the dictionary
pairs¼ line.split()
pwDict.update({pairs[0] : pairs[1]})

fp.close()
except:
print'File Handling Error'
fp.close()

# Once all the files have been hashed # I calculate the elapsed time

elapsedTime ¼ time.time() - startTime print'Elapsed Time:',
elapsedTime,'Seconds' # print out a few of the dictionary entries # as an example
print'Passwords Generated:', len(pwDict) print
cnt ¼ 0
for key,value in (pwDict.items()):

print key, value
cnt +¼ 1
if cnt > 10:

break;
print
# Demonstrate the use of the Dictionary to Lookup # a password using a known
hash value
pw ¼ pwDict.get('c6f1d6b1d33bcc787c2385c19c29c208') print'Hash Value
Tested ¼ \ 2bca9b23eb8419728fdeca3345b344fc'
print'Associated Password¼ '+ p w
# Program Output
Elapsed Time: 50.504999876 Seconds Passwords Generated: 5399020

3e47ac3f51daffdbe46ab671c76b44fb K23IH
a5a3614f49da18486c900bd04675d7bc $@fL1
372da5744b1ab1f99376f8d726cd2b7c hGfdd
aa0865a47331df5de01296bbaaf4996a 21ILG
c6f1d6b1d33bcc787c2385c19c29c208 #I#$$
c3c4246114ee80e9c454603645c9a416 #$b$g
6ca0e4d8f183c6c8b0a032b09861c95a L1H21
fd86ec2191f415cdb6c305da5e59eb7a HJg@h
```

335ef773e663807eb21d100e06b8c53e a$HbH
d1bae7cd5ae09903886d413884e22628 ba21H
a2a53248ed641bbd22af9bf36b422321 GHcda

Hash Value Tested¼ 2bca9b23eb8419728fdeca3345b344fc Associated Password¼ #I#$$

As expected, the multi-core approach improves the performance and the solution is still straightforward and easily readable.
Reviewing the results of the single and multi-core implementations of the Rainbow Tables we find the following.

Implementation
Rainbow Processing generator time (s) Passwords

per second Notes

Single core 89.11
solution
Multi-core solution 50.50
60 K

106 K Typical implementation today processing using a single core Use of Pool Class simplifying implementation

In addition to the performance results, the output of the generated key value pairs (MD5 hash and associated password are written to a file) and can be used later as part of a larger repository. Since the outputs are simple text files as shown in Figure 10.1 they are easily readable.

SALTED Password MD5 HASH Associated Password
17ae80e34251ad4e2a61bc81d28b5a09 aa
9e6e21b8664f1590323ecaae3447ebae ab
e708b3b343cfbc6b0104c60597fff773 ac
dcea12b90e71b523db8929cf6d86e39d ad
70a3c3c78c79437dfc626ff12605ccb3 ae
41afc8d925b0b94996035b4ef25346ec af
14b45c3a22caf6c2d5ed2e3daae152ee ag
3c4493267b90a86303c1d30784b8f33b ah

```
5cbba376bbb10688ece346cbfa9b8c28 aG
ffad7a79a9b2c646ef440d1447e18ebe aH
…
…
eba8905561738d699d47bd6f62e3341c eeg2
13605b276258b214fdafccac0835fc67 eeg3
0e3a4e8aded5858c94997d9f593c4fa3 eeg!
a6ab3da0c0cc7dc22d2a4cfe629ba0f2 eeg@
8f005cf0ea4b2bf064da85e1b2405c00 eeg#
60c797433e1f1ff9175a93373cf1a6ff eeg$
e847612d8cab3184caa4d120c212afec eeha
75cc13015bd40e2a6da4609458bfbe01 eehb
```

FIGURE 10.1
Plaintext Rainbow Table output abridged.

CHAPTER REVIEW

In this chapter, I began the process of addressing multi-core processing using Python by leveraging the cores already available on your laptop or desktop system. I introduce a couple of different approaches to multiprocessing and provide an overview of the Python Standard Library multiprocessing. I also examined two common digital investigation mainstay functions that can benefit from multiprocessing: (1) File Hashing and (2) Rainbow Table generation.

In Chapter 11, I will demonstrate how you can take this approach into the cloud and leverage cloud services that could offer 10, 50, 100, or even a 1000 cores to expand the horizon of multiprocessing from the desktop to the cloud.

SUMMARY QUESTION/CHALLENGE

1. What other digital investigation or forensic applications do you believe could benefit from multiprocessing?
2. What are some of the key elements when designing multiprocessing solutions?
3. What is the best defense today against Rainbow Table-based password attacks?
4. The Rainbow Table example is limited by resources and will fail when the program runs out of memory. How would you modify the program such that it could continue to generate password/hash combinations in the face of memory

limitations?

Additional Resources

http://www.net-security.org/article.php?id ¼1932&p¼1.
http://docs.python.org/2/library/itertools.html#module-itertools.
http://docs.python.org/3.4/library/multiprocessing.html#module-multiprocessing.

CHAPTER

Rainbow in the Cloud 11

CHAPTER CONTENTS

Introduction 289
Putting the Cloud to Work 289
Cloud Options 290
Creating Rainbows in the Cloud 292
Single Core Rainbow 295
Multi-Core Rainbow 296
Password Generation Calculations 300
Chapter Review 302
Summary Question/Challenge 303
Additional Resources 303

INTRODUCTION

One of the significant advantages of building applications in Python is the ability to deploy applications on virtually any platform, including of course the cloud (see Figure 11.1). This not only means that you can execute Python on cloud servers, but you can launch them from any device you have handy, desktop,

laptop, tablet, or smartphone.

PUTTING THE CLOUD TO WORK

In Chapter 10, I created a simple application that searches a dictionary for certain words. I created a single core application and then one that leverages multiple cores. As you can see in Figure 11.2, I have executed the two programs that perform the simple dictionary search program. I executed them from my iPad with the application running in the cloud. This code is unchanged from the version created in Chapter 10:

sp.py is the single processing version
mp.py is the multiprocessing version

In Figure 11.3, I execute both the Simple Single Core and Multi-Core application from my desktop browser. As you can see the Multi-Core runs a bit faster even though I am only running the multiprocessing application on two-core cloud computers at Python Anywhere.

© 2014 Elsevier Inc. All rights reserved.

FIGURE 11.1 Typical cloud configuration.

Note that in these examples I have not changed a line of code to execute the Python code in the cloud. Since the code is executed by a standard Python

interpreter and I have used Python Standard Libraries the code just runs. If you utilize third-party libraries then you will have to get those added to the cloud-based Python installations (this is not impossible to do), but it is much easier if we stick with the Standard Library. I am using the cloud service Python Anywhere or www.pythonanwhere.com as shown in Figure 11.4. Python Anywhere is a terrific place to start experimenting with Python applications in the cloud. Signup is free for a minimal account, giving you access to a couple of cores and storage for your programs. I have a $12/month plan and this gives me 500 GB of storage and 20,000 CPU seconds allowed per day. The seconds are seconds of CPU time—so any time that any processor spends working on your code, internally it is measured in nanoseconds. Most online cloud services reference the Amazon definition of the CPU second which is: “the equivalent CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon 64 bit processors.”

CLOUD OPTIONS

Many cloud options exist that allow access to as few as 2 and as many as 1000 cores on which to execute. Table 11.1 provides an overview of just a few that you could investigate further (note, some are ready to run Python code, others require you to set up your own Python environment). In addition, some have a custom application interface or application interface (API) that you must use for multiprocessing, while others use a native approach.

FIGURE 11.2
Cloud execution from iPad.

FIGURE 11.3
Desktop execution of the simple and multiprocessing Python applications executing in the cloud.

FIGURE 11.4
Python Anywhere Home Page.

Table 11.1 A Few Python Cloud Options Cloud Service URL Notes
[PythonAnywhere] pythonanywhere.com
[PiCloud] picloud.com
[Digital Ocean] digitalocean.com
Others Amazon, Google, ATT,

IBM, Rackspace just to mention a few
Runs native Python code for version 2.6, 2.7, 3.3
Figure 11.5
Runs native Python code, but requires the import of their cloud module for

multiprocessing.
See Figures 11.6 and 11.7
Requires you to install a Python package for the environment and your application. See Figures 11.8 and 11.9
As you expand your applications these services offer a variety of solutions

CREATING RAINBOWS IN THE CLOUD

There are considerable trade-offs and options available for creating high performance Python-based investigation platforms. One of the interesting applications that is suited nicely for cloud execution is the generation of Rainbow Tables discussed and experimented with in Chapter 10. Moving both the single and multiprocessing versions of this application to the cloud requires some considerations. First, I have decided to use Python Anywhere to demonstrate this capability. As I mentioned earlier, it is a great way to get started in the cloud, because the environment executes native Python applications for version 2.6, 2.7, and 3.3. Since I have made sure to only use Standard Library modules and core language elements, portability to the

FIGURE 11.5

Python Anywhere Plans.

FIGURE 11.6
PICloud Home Page.

PiCloud

Platform Pricing Solutions Sign up

Home — Pricing

Public Cloud

Signing up gives you instant access to our public cloud solution. Your workloads run on our cloud, which means you get on-demand access to as many cores as you need. If you're looking to use PiCloud with private infrastructure, see our private cloud offering.

Computation

You pay only for what you use. There is no minimum fee. We charge you for the number of core hours your functions consume. Each of your functions can be run on any of the cores we offer. *Please note, that while our pricing is expressed in hours for clarity, your actual charge will be prorated by the millisecond.*

Core Type	Compute Units [1]	Memory	Disk	Max Multicore [2]	Price/Hour
c1 (default)	1	300 MB	15 GB	1	$0.05
c2	2.5	800 MB	30 GB	8	$0.13
f2	5.5 w/ HT	3.7 GB	100 GB	16	$0.22
m1	3.25	8 GB	140 GB	8	$0.30
s1 [3]	0.5 to 2	300 MB	4 GB	1	$0.04

[1] A **compute unit** as defined by Amazon provides "the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor." All of our cores have 64-bit architectures.

FIGURE 11.7
PICloud Plans.

FIGURE 11.8
Digital Ocean Home Page.

Monthly	Hourly	Memory	CPU	Storage	Transfer	Get Started
$160	$0.238	16GB	8 Cores	160GB SSD	6TB	Sign Up
$320	$0.476	32GB	12 Cores	320GB SSD	7TB	Sign Up
$480	$0.705	48GB	16 Cores	480GB SSD	8TB	Sign Up
$640	$0.941	64GB	20 Cores	640GB SSD	9TB	Sign Up
$960	$1.411	96GB	24 Cores	960GB SSD	10TB	Sign Up

FIGURE 11.9
Digital Ocean Plans.

Python Anywhere cloud is pretty simple. However, I am going to make a couple of significant changes to the experimental code that I developed in Chapter 10:

(1) Minimize the memory usage within the programs by eliminating the use of the lists and dictionaries.
(2) Simplify the characters used to keep the resulting password generation reasonable
(3) Expand the generated password lengths to 4- to 8-character passwords

The resulting code listings for both the single and multiprocessing versions are shown here:

Single Core Rainbow

```
# Single Core Password Table Generator
# import standard libraries

import hashlib # Hashing the results
import time # Timing the operation import sys
import os
import itertools # Creating controled combinations

#
# Create a list of characters to include in the # password generation
#

chars ¼ ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

```
# Define a hypothetical SALT value
SALT¼"&45Bvx9"
# Define the allowable range of password length PW_LOW¼4
PW_HIGH¼8

print'Processing Single Core'
print os.getcwd()
print'Password Character Set:', chars
print'Password Lenghts:', str(PW_LOW),'-', str(PW_HIGH)

# Mark the start time startTime¼time.time()
# Open a File for writing the results

try: # Open the output file
fp¼open('PW-ALL','w')

except:
print'File Processing Error' sys.exit(0)

# create a loop to include all passwords # within the allowable range
pwCount¼0
for r in range(PW_LOW, PW_HIGH+ 1):
#Apply the standard library interator
for s in itertools.product(chars, repeat¼r):
# Hash each new password as they are # generated

pw ¼".join(s) try:
md5Hash¼hashlib.md5()
md5Hash.update(SALT +pw)
md5Digest¼md5Hash.hexdigest()

# Write the hash, password pair to the file fp.write(md5Digest +"+pw +'\n')
pwCount+¼ 1
del md5Hash

except:
print'File Processing Error'
# Close the output file when complete fp.close()
```

When complete calculate the elapsed time elapsedTime¼time.time() - startTime print'Single Core Rainbow Complete' print'Elapsed Time:', elapsedTime print'Passwords Generated:', pwCount print

Multi-Core Rainbow
Multi-Core Password Table Generator
import standard libraries

import hashlib # Hashing the results import time
import os
import itertools
import multiprocessing # Timing the operation

Creating controled combinations # Multiprocessing Library

Create a list of characters to include in the # password generation
#

chars¼ ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
Define a hypothetical SALT value SALT¼"&45Bvx9"
Define the allowable range of password length
PW_LOW¼4 PW_HIGH¼8
def pwGenerator(size):

pwCount ¼0
create a loop to include all passwords # within range specified

try:
# Open a File for writing the results fp¼open('PW-'+str(size),'w')
for r in range(size, size+ 1):
#Apply the standard library interator

for s in itertools.product(chars, repeat¼r): # Process each password as they are # generated
pw¼'.join(s)

# Perform hashing of the password md5Hash¼hashlib.md5()
md5Hash.update(SALT +pw)
md5Digest¼md5Hash.hexdigest()

```
# Write the hash, password pair to the file fp.write(md5Digest +"+pw +'\n')
pwCount+¼ 1
del md5Hash

except:
print'File/Hash Processing Error'
finally:
fp.close()
print str(size),'Passwords Processed¼', pwCount

#
# Create Main Function #

if __name__ ¼¼'__main__':

print'Processing Multi-Core'
print os.getcwd()
print'Password string:', chars
print'Password Lengths:', str(PW_LOW),'-', str(PW_HIGH)

# Mark the starting time of the main loop startTime¼time.time()
#create a process Pool with 5 processes
corePool¼multiprocessing.Pool(processes¼5)
#map corePool to the Pool processes
results¼corePool.map(pwGenerator, (4, 5, 6, 7, 8))
elapsedTime¼time.time() - startTime
# When complete calculate the elapsed time

elapsedTime¼time.time() - startTime print'Multi-Core Rainbow Complete'
print'Elapsed Time:', elapsedTime print'Passwords Generated:', pwCount print
```

You can see the results of executing both the single core and multi-core solutions on the Python Anywhere Cloud, respectively, in Figures 11.10 and 11.11. I have also included the execution from my Linux box in Figure 11.12. The actual performance will vary of course based on many factors. Since my Linux box is dedicated and is running a quad-core processor at 3.0 GHz it outperforms the cloud service. The single core and multi-core results demonstrate a proportional result with the multi-core solution outperforming the single core solution as expected (see Table 11.2).

FIGURE 11.10
Python Anywhere Single Core Execution Results.

FIGURE 11.11
Python Anywhere Multi-Core Execution Results.

```
chet@PythonForensics: ~/Desktop/Rainbow Test
chet@PythonForensics:~/Desktop/Rainbow Test$ python SingleCoreRainbow.py
Processing Single Core
/home/chet/Desktop/Rainbow Test
Password string:  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
Password Lenghts:  4  -  8
Single Core Rainbow Complete
Elapsed Time:  80.9314088821
Passwords Generated:  19173376

chet@PythonForensics:~/Desktop/Rainbow Test$ python MultiCore\ Rainbow.py
Processing Multi Core
/home/chet/Desktop/Rainbow Test
Password string:  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
Password Lenghts:  4  -  8
4  Passwords Processed=  4096
5  Passwords Processed=  32768
6  Passwords Processed=  262144
7  Passwords Processed=  2097152
8  Passwords Processed=  16777216
Multi Core Rainbow Complete
Elapsed Time:  63.3727021217

chet@PythonForensics:~/Desktop/Rainbow Test$
```

FIGURE 11.12
Standalone Linux Single/Multi-Core Execution Results.
Table 11.2 Summary of Execution Results Execution
configuration Passwords generated Time to and processed process(s) Passwords per second

Standalone Quad Core Linux
Single Core
Multi-Core

Python Anywhere Single Core
Multi-Core
19,173,376 80.93 19,173,376 63.37 236,913 302,562

19,173,376 210.99 19,173,376 142.93 90,873 134,145

PASSWORD GENERATION CALCULATIONS

One question you might be asking at this point is how many unique combinations of passwords are there? In order to be reasonable, let us start with the number of possible 8-character passwords by just using lowercase letters. The answer is shown in Figure 11.13 calculated by elPassword [elPassword]. In

Figure 11.14, elPassword calculates the number of unique 8-character passwords using upper and lowercase letters, numbers, and special characters.

FIGURE 11.13
elPassword 8-character combinations of lowercase letters.

FIGURE 11.14
elPassword 8-character full ASCII character set.

Using the online resource from LastBit, [LastBit] along with our best performance of 302,000 passwords per second, we can calculate the length of time required for a brute force attack. In Figure 11.15–11.18, I performed four separate runs. The first two using all lowercase characters with 1 and 100 computers, and the last two using the full ASCII set with 100 and 10,000 computers, respectively. Try them out for yourself.

FIGURE 11.15
Last Bit calculation lowercase using 1 computer.

FIGURE 11.16
Last Bit calculation lowercase using 100 computers.

FIGURE 11.17
Last Bit calculation ASCII set using 100 computers.

FIGURE 11.18
Last Bit calculation ASCII set using 10,000 computers.

CHAPTER REVIEW

In this chapter I introduced Python Anywhere, a cloud service that runs native Python code in the cloud. I demonstrated how simple it is to run native Python code in the cloud from multiple platforms. I then modified the Rainbow Table password generator to minimize memory usage, reduced the characters, and expanded the generation

to include password lengths of 4- to 8-character solutions. I then examined the performance of each result to determine how long it would take both on a higher performance Linux platform and in the cloud. I took those results and extrapolated them in order to determine the time and computers needed to reasonably crack 8-character passwords.

SUMMARY QUESTION/CHALLENGE

1. What other applications would be useful to execute within cloud environments that would benefit the forensic and investigation community?
2. As of this writing, Intel and AMD are experimenting with single processor core counts of 16, 32, 64, and 96 cores. If we extrapolate just a few years in the future it would not be unreasonable to expect the achievement of 1000-core CPUs. How will this change our ability to generate and hash passwords, crack encryption, or search data?
3. Develop and test your own multi-core solution on your desktop environment. Design them such that they can be easily deployed to a cloud platform like Python Anywhere. Establish a free account on Python Anywhere and experiment with your single and multi-core solutions.

Additional Resources

http://www.pythonanywhere.com . http://www.picloud.com.
http://www.digitalocean.com.
http://projects.lambry.com/elpassword/. http://lastbit.com/pswcalc.asp.

CHAPTER

Looking Ahead 12

CHAPTER CONTENTS

Introduction 305
Where Do We Go From Here? 307
Conclusion 312
Additional Resources 312

INTRODUCTION

The world of digital investigation and computer forensics is approaching 25 years in age. I had the privilege of working with Ron Stevens from the New York State Police who was one of the early pioneers and one of the first troopers in the country to successfully collect evidence from computers and use that evidence to convict those involved in criminal activity. “As one of the first law enforcement agencies to respond to the threats posed by the techno-criminal, the New York State Police launched the Computer Crime Unit in 1992” (Stevens, 2001).

I also had the privilege of serving as the primary investigator on the first digital forensic research effort sponsored by the Air Force Research Laboratory in 1998. The effort was conceived by two pioneers in the field working at the U.S. Air Force Research Lab, Joe Giordano and John Feldman. The effort was entitled: Forensic Information Warfare Requirements and resulted in a comprehensive report that outlined a set of requirements for the field moving forward. (FIW Report, 1999)

However, it would be a mistake to think of digital investigation, cybercrime investigation, or computer forensics as a mature science or discipline today. The following is an excerpt from a presentation I gave at the First Digital Forensic Research Workshop in 2001 (DFRWS, 2001).

Digital Evidence Fundamentals

– Digital evidence is vast, complicated, and can be easily manipulated, concealed, destroyed, or never found
– Connecting the dots can be an arduous process, fraught with uncertainty and in many cases is inconclusive
– The distributive nature of cyber-crime and cyber-terrorism makes tracing the perpetrators, the victims, and the technology used to execute the attack or crime difficult
– Our results will likely be challenged, thrown out, ignored, misunderstood, and constantly questioned

© 2014 Elsevier Inc. All rights reserved.

Cyber-forensic technologies must be:

– Reliable
– Accurate
– Nonreputable
– Easy to use
– Foolproof
– Secure
– Flexible
– Heterogeneous
– Distributed
– Automatic
– Free, or at least cheap

Fundamental questions—Cyber-forensic technologies

– Who collected the digital evidence?
– With what tool or technology?
– By what standard or practice is this based?
– Who audits and validates the practices?
– How are the identities of the digital detectives bound to the digital evidence?

– How was the evidence handled once identified?
– How is the evidence validated, and by whom?
– For how long is it valid?
– How is it stored and secured?
– How is the integrity of digital evidence assured?
– What technology was used to assure it?
– Why do I trust the tool or technology?

– Who developed it?
– Where and under what conditions was it developed?
– What underlying software and hardware does the technology rely on?
– Who validated or accredited the technology and the process?
– Which version(s) are accredited?
– Who trained and accredited the users

– Is the evidence distinctive?
– Is the evidence privileged?
– Is the evidence corroborated?

– When was the file created, modified, or destroyed?
– When was the transaction executed?
– When was a message transmitted or received?
– When was the virus or worm launched?
– When was the cyber-attack initiated?
– How long after the reconnaissance stage was completed did the attack commence?
– In what time-zone?
– At what point was the system log still valid?
– Did the suspect have the opportunity to commit the crime?

Fundamental questions

– Where is the suspect in cyber-space?
– How can I trace his or her steps?
– Technically
– Legally
– Where are they likely to strike next?
– Are they working with accomplices or insiders?
– What capabilities do they posses?

– Bandwidth
– Computing power
– Savvy
– Resources
– Have we seen them before?
– Are they more sophisticated now vs. a year ago?
– Who are their accomplices?

Summary

– We must focus our attention on identifying and addressing the key fundamental elements of digital forensics and digital evidence
– We must work together to build technologies that address the issues surrounding these fundamental elements
– We need to perform research where the goal is to use the science of forensics to help answer the harder questions surrounding cyber-criminals and terrorists
– Their location, sophistication, likely next targets
– What do we need to do to stop them?
– How do we improve our defenses against them?

The scary part about this short trip down memory lane is that I could give this presentation today, and it would still be for the most part relevant. The big question is where we go from here and how will Python Forensics play a role?

WHERE DO WE GO FROM HERE?

This book has identified key areas where research and development of new solutions are possible when combining core challenge problems with the Python language. It could certainly be argued that other languages could produce viable solutions to these challenges. However, the questions are:

– Would these alternative solutions be open source and free?
– Would they be cross platform ready (Windows, Mac, Linux, Mobile, and in the Cloud)?
– Would they be accessible and readable by anyone?
– Would they have the global worldwide support?
– Do they support a collaborative environment that would provide an on-ramp for computer scientists, social scientists, law enforcement professionals, or students new to the field?

The next big steps from my view are as follows:

(1) Creation of a true collaborative environment where people can share information (challenge problems, ideas, and solutions).
(2) Obtain nonintimidating support as technology is advanced in support of new investigative challenges.
(3) Development of a repository of programs and scripts that could be downloaded, applied to real word problems, expanded, and improved.
(4) Integration with on-demand training courses to dive deeply into core areas of Python and forensics.
(5) A validation/certification process that would allow third-party organizations (such as NIST) to validate Python supplied solutions for use by law enforcement. Once validated they could be safely utilized on real cases. Once the process has been created much of the validation work could be automated through the use of standardized forensic test images to accelerate the validation process.
(6) Vendors could supply application interfaces that open the possibility to integrate new Python-based solutions into existing forensic technologies. This would actually improve the capabilities of vendor solutions, allow them to address new issues in a more timely fashion, and make their products more valuable in the marketplace.
(7) The creation of a cloud-based experimental platform that has thousands (or even hundreds of thousands) of processor cores, petabytes of storage, and terabytes of memory that could be applied and harnessed to solve computationally difficult problems. In addition, this environment would be open to academia and students in order to rapidly advance classroom problems. Collaboration across universities, colleges, practitioners, vendors, and researchers to solve truly hard problems. The environment could provide a competition surface for new innovations, and teams could compete on the national and international stage. Benchmarks for specific solution types could be created in order to understand the performance characteristics of various solutions.
(8) Key challenge problems for consideration:
Throughout this book, I provided Python-based examples for several key areas. These examples provide the basics, however, much more work is needed. I have identified some of the key challenges that require additional work and focus.
a. Advanced searching and indexing: search and indexing are certainly core elements during any investigation. However, improving the speed, accuracy, and relevance of search and index results is necessary. Investigators need solutions

that can provide the following in a timely fashion: i. Search and index results that deliver information that is relevant to their case. This rich search/index capability must uncover information that is not obvious or is missed by present-day technologies. For example, search results that include time and spatial connections into clear view.

ii. Search and index results that connect information from multiple cases together, identifying connections between accomplices, Internet, phone, time, location, and behavioral analysis that were previously unconnected.

b. Metadata extraction: Images and multimedia content contain a plethora of metadata, including but not limited to time, date, the device they were created on, location information, subject content, and much more. The extraction, connection, and reasoning about this information is today left to the investigator. New innovations bring the promise of rapidly extracting and connecting this information to provide a broader more comprehensive view of the crime scene.

c. Event synchronization: According to Statistic Brain (2014) in 2013 there were an average of 58 million tweets per day from over 645 million registered Twitter users. In addition, (Statistic Brain, 2014) 5.9 Billion Google searches per day and over 2 trillion searches for the year 2013 were recorded. This is only a fraction of the total Internet events that occur each day and each year. Our ability to synchronize and reason about events whether from the Internet, a corporate network or even an individual desktop may seem beyond comprehension. We need to develop new innovations that can make sense, isolate, and provide conclusive proof about such actions and events.

d. Natural language: The Internet has certainly broken down boundaries and provided interaction and communication instantly across the globe. As I just scratched the surface of Natural Language Processing (NLP) in Chapter 7, you can see the potential power of being able to process language. The application for NLP for the extraction of meaning, determining authorship, deciphering intent are all within our grasp. Expanding these technologies to process a wide range of languages, improving deductive reasoning, extracting persons, places and things, and assessing likely past, present or future actions are possible.

e. Advancement in Python. The printed examples and source code provided in this book were developed for Python 2.7.x in order to ensure the broadest compatibility across computing platforms. However, all the examples will also be available online for download with solutions for 2.7.x and 3.3.x. Python and third-party Python libraries continue to expand at the speed of the Internet. Some even claim Python will be the last programming language you will ever have to learn. I think this is a bit overstated however, some of the attributes of the language will be fundamental to future languages. At the time of this writing

February 9, 2014 to be exact, Python Version 3.4 was released. According to the Python Programming Language Official Web (www.python.org), the major language improvements included: 1. a “pathlib” module providing object-oriented file system paths 2. a standardized “enum” module
3. a build enhancement that will help generate introspection information for built-ins
4. improved semantics for object finalization
5. adding single-dispatch generic functions to the standard library
6. a new C API for implementing custom memory allocators
7. changing file descriptors to not be inherited by default in subprocesses
8. a new “statistics” module
9. standardizing module metadata for Python’s module import system 10. a bundled installer for the pip package manager
11. a new “tracemalloc” module for tracing Python memory allocations 12. a new hash algorithm for Python strings and binary data 13. a new and improved protocol for pickled objects
14. a new “asyncio” module, a new framework for asynchronous I/O As you can see the language evolution continues at a brisk pace along with the development of new and improved third-party Python libraries that help accelerate forensic and digital investigation tools. There are so many third-party libraries and tools, I decided to list just my top 10.
1. Pillow. A new library that builds off of the more traditional Python Image Library for those that process and examine digital images.
2. wxPython. For those of you that need to build cross platform graphical user interfaces this is my preferred toolkit.
3. Requests. One of the best http interface libraries.
4. Scrapy. If your forensics investigations require you to perform web scraping, this library will help you build new and innovative approaches.
5. Twisted. For those needing to develop asynchronous network applications.
6. Scapy. For those who perform packet sniffing and analysis, scapy provides a host of features that can speed your development.
7. NLTK. Natural Language Toolkit—This toolkit is vast and for those investigating text and language constructs, it is a must.
8. IPython. When experimenting with new language elements, libraries, or modules this advanced Python Shell will assist you in every aspect of your work or struggles.
9. WingIDE. This is not a library, but this Integrated Development Environment gets my vote for the best IDE. The Professional version provides even the most savvy researcher with the tools they need. 10. Googlemaps. Many forensic

applications collect geographically tagged information, this package allows you to easily integrate with the Google mapping system.

f. Multiprocessing: In order to effectively attack any of these areas, our ability to access the power of the latest processors and cloud-based solutions is essential (Figure12.1). “The International Technology Roadmap for Semiconductors 2011,” a roadmap forecasting semiconductor development drawn up by experts from semiconductor companies worldwide, forecasts that by 2015 there will be an electronics product with nearly 450 processing cores, rising to nearly 1500 cores by 2020 (Heath).

What is available today from the two major processor manufactures, Intel® and AMD® are depicted in Figures 12.2 and 12.3. For under $1000 you can own these processors and for under $3000 you can build a system with two processors yielding 20-32 cores, 64 GB of memory and a couple of terabytes of storage. This is certainly a step toward multicore and multiprocessing solutions, and when coupled with well-designed multicore

FIGURE 12.1
Multiprocessing in the Cloud.

FIGURE 12.2
AMD 6300 Series 16 Core Processor.

FIGURE 12.3
Intel Xeon E7 Series 10 Core 20 Thread Processor.

Python applications could take a giant step toward advancing the state-of-theart. Once we see these production solutions produce 64, 128, 256, and 1024 cores on a single processor, the world will change once again.

CONCLUSION

Applying the Python language to digital investigation and forensic applications has great promise. What is needed is a collaborative community that includes: practitioners, researchers, developers, professors, students, investigators, examiners, detectives, attorneys, prosecutors, judges, vendors, governments, and research institutes. In addition, a cloud-based computing platform with thousands of cores, petabytes of storage, and terabytes of memory is necessary.

I challenge you to participate. If everyone reading this book would submit one idea, challenge problem or solution that would be a tremendous start. Visit Python-Forensics.Org to get started.... I would love to hear from you!

Additional Resources

Fighting Cyber Crime hearing before the subcommittee on Crime of the Committee on the Judiciary, http://commdocs.house.gov/committees/judiciary/hju72616.000/hju72616_0f.htm; 2001.

Forensic Information Warfare Requirements Study F30602-98-C-0243. Final technical report, February 2, 1999, Prepared by WetStone Technologies, Inc.
A road map for digital forensic research, Utica, New York, http://www.dfrws.org/2001/dfrwsrm-final.pdf; August 7–8, 2001.
January 1, 2014, http://www.statisticbrain.com/twitter-statistics/.

Cracking the 1,000-core processor power challenge. Nick Heath, http://www.zdnet.com/ cracking-the-1000-core-processor-power-challenge-7000015554/.

Index

Note: Page numbers followed by b indicate boxes, f indicate figures and t indicate tables.

A

Advanced searching and indexing, 308
Advancement in Python, 309
AMD 6300 Series 16 Core Processor, 311–312, 311f

B

Built-in constants
Boolean values, 31
hard core developer, 31–32
strongly typed language, 31–32, 31f variables, 31

Built-in exceptions, 33–34
Built-in functions
hex( ) and bin( ), 27–28, 27f Python 2.7, 30, 30t
range( ), 28–30, 28f
Built-in types
bitwise operations, 32
categories, 32

C

Class Logging, 141
Cloud-based experimental platform, 308
Cloud-based Python environments

cloud execution, iPad, 289, 291f
desktop execution, Python applications,

289 , 291f
Digital Ocean Home Page, 294f

Digital Ocean Plans, 294f
elPassword 8-character combinations of

lowercase letters, 300, 300f
elPassword 8-character full ASCII character set,
300, 301f
execution results table, 300–301
Last Bit calculation ASCII set, 301, 302f Last Bit calculation lowercase, 301,
301f, 302f
Multicore Rainbow, 291f, 296–299
PICloud, 293f, 294f
Python Anywhere (see Python Anywhere) Python Standard Libraries, 290
Rainbow Table generator (see Rainbow
Table generation)
simple single core, 289, 291f
Single Core Rainbow, 295–296
Standalone Linux Single/Multi Core Execution,
298, 299f
Standard Library modules, 292–295
typical cloud configuration, 289, 290f Code examination
Command line parser, 149–150
Comma separated value (CSV) Writer class,
150–151
EXIF and GPS processing, 144–148
Logging Class, 148–149
Code walk-through, Python forensics
ParseCommandLine, 101–102
PrintBuffer function, 106–107
p-search functions, 101
SearchWords function, 100–101, 103–107 standard library, 100
ValidateFileRead(theFile), 103
Command line parser, 141, 149–150
Corpus
challenges, 184–185
the Internet, 193–194
NLP, 184
working with, 185–186
Cybercrime investigation
cost and availability, tools, 3, 4f

data vs. semantics, 4, 5f
Linux computer, 2
next-generation investigator, 4, 5f
smart mobile devices, 2
technology developers and investigators,
2–3, 3f
Cyber-forensic technologies, 306–307

D

Data vs. semantics
description, 4
knowledge and process, 4
targets/hotspots, 4

The Daubert standard, 9–10
Digital evidence fundamentals, 305–306 Digital Ocean

Home Page, 294f
313
E

Event synchronization, 309
The Exchangeable Image File Format (EXIF) GPSTAGS, 133–137
TAGS, 131–133
Exclusive OR (XOR)
application, 32–33, 33f
evaluation, 33
GMT, 166
GPS signals, 168–169
Harrison H1 clock, 165–166, 166f history, 168, 168f
IETF, 167
module, 169–173
NTP, 173
precision and reliability, 167 UTC, 166

F

FCC. See Federal Communications Commission (FCC)
Federal Communications Commission (FCC), 126

File hashing
forensic tools, 272
multicore solution A, 274–277
multicore solution B, 277–279
single core solution, 272–273
Forensic evidence extraction
Class Logging, 141
code examination, 141–151
Command line parser, 141
cvsHandler, 141
desktop tools, 125–126
dictionary creation, 129
digital data structures, 125–126
digital photographs, 126
EXIF and GPS Handler, 141
FCC, 126
Full code listings, 151–158
hash value, 129
iteration, dictionary, 130
library mapping, 137, 139t
Lists and Sets structures, 128
p-gpsExtractor context diagram., 137, 140f PIL Test-Before Code, 131
p-ImageEvidenceExtractor Fundamental Requirement, 137
“Porsche”, 128–129
printing, sorted dictionary, 129
program execution, 158–159
simple key and value extraction, 130
social network applications, 126
WingIDE project, 137, 140f
Forensic log
CSV file output, 261, 261f, 262f
TCP capture, 260–261
UDP capture, 261
Forensic Log file, 159, 163f
Forensic time
atomic clocks, 168
documents and timestamps, 166–167

G

Global positioning system (GPS)
EXIF GPSTAGS, 133–137
geolocation information, 134
Python language elements, 159

GMT. See Greenwich Mean Time (GMT) GPS. See Global positioning system (GPS) Greenwich Mean Time (GMT)

and UTC, 171
world's official time, 168
H
HashFile, 71–74
I

IDE. SeeIntegrated development environments (IDE) IDLE, 38–39, 38f
IETF. See Internet Engineering Task Force (IETF) Integrated development environments (IDE)

description, 37
IDLE, 38–39
Ubuntu Linux, 42–46
WingIDE, 39–42

Intel Xeon E7 Series 10 Core 20 Thread Processor, 311–312, 311f
Internet Engineering Task Force (IETF), 167
IOS python app
Apple App Store page, 47, 48f
SHA256 Hash, 47
Shell, 46–47

M

Metadata extraction, 309
Mobile devices
iOS python app, 46–48
Windows 8 phone, 49–50
Multi Core Execution, 298, 299f
Multicore Rainbow, 296–299
Multiprocessing, forensics
central processing units (CPUs), 266 digital investigation mainstay functions,

288 File Hash, 272–279
File Search Solution, 270–272
Hash Table Generation, 279–287
law enforcement agencies, 265
modern multicore architectures, 266 multicore processing methods, 269 Python code, 269
Python Multiprocessing Support, 266–269 Python Standard Library, 266
SearchFile function, 269
Single Core File Search Solution, 270 Windows environment, 266
Multiprocessing in the Cloud, 311–312, 311f

N

Natural Language Processing (NLP)
“Computing Machinery and Intelligence”, 184 corpus (see Corpus)
definitions, 183–184
dialog-based systems, 184
forensic applications, 202
IITRI, 183–184
NLTK (see NLTK library)

Natural Language Toolkit (NLTK), 36–37 Network forensics
interactive scanning and probing, 237–238 investigation, 205
network investigation methods, 205–206 packet sniffing, 238–240
ping sweep (see Ping sweep)
port scan (see Port scan)
professional tools and technologies, 205 program execution and output, 259–261
programs, 235
PSNMT, 247–249
raw sockets, Python, 240–247
sockets (see Sockets)
third-party modules, 235
Network time protocol (NTP)
description, 173
library, 174–177
Next-generation cyber warrior, 4, 5f
NLTK. See Natural Language Toolkit (NLTK)
NLTK library
description, 185

experimentation
from __future__ import division, 187 print len(rawText), 188
print len(tokens), 188
print len(vocabularyUsed), 189
print newCorpus.abspaths(), 187
print newCorpus.fileids(), 187
print sorted(set(textSimpson)), 189 print textSimpson.collocations(), 190
print textSimpson.concordance(myWord), 190, 191
print textSimpson.similar(myWord), 191
print tokens[0:100], 188
print type(newCorpus), 187
print type(textSimpson), 189
simpsonVocab.items(), 192
simpsonVocab ¼ textSimpson.vocab(), 192 NLTKQuery application (see NLTKQuery application)
operations, 186t
NLTK.org Installation url., 185f
NLTKQuery application
capabilities, 202
_classNLTKQuery.py, 196–198
execution trace, 199–202
NLTKQuery.py, 195–196
_NLTKQuery.py, 198–199
source files, 194
NTP. See Network time protocol (NTP) ntplib
decompressed ntplib-0.3.1, 176f
European NTP pool project, 177–179, 178f installation, 175, 176f
NIST time servers, 177, 178f
ntplib-0.3.1.tar.gz., 175f
Python download page, 174f
third-party source, 177
verification, installation, 175, 177f

O

One-way file system hashing
best-use cases, 57–58
characteristics, 56
cryptographic hash algorithms, 57, 57t tradeoffs, 57

Operating systems (OS), 35–36
OS. See Operating systems (OS)
P

Packet sniffing
port scanning, 238
SPAN port connections, 238–240, 239f

ParseCommandLine( )
argparse, 102, 102f
command line options, 66–67
global variables, 68
program execution, 67
p-search command line argument, 101, 102t sha384, 67
ValidateFileName(), 101

p-gpsExtractor
coordinates, program execution, 159, 160f Forensic Log file, 159, 163f
Map zoom into Western Europe, 159, 161f Map zoom to street level in Germany,

159 , 162f
online mapping program, 158
Results.csv file snapshot, 159, 162f
Windows execution screenshot, 158, 159f

PICloud
Home Page, 293f
PIL. See Python Image Library (PIL)
Ping sweep
application, 211
execution, 224–225
GUI environment, 211
guiPing.py code, 218–224
ICMP, 211
ping.py, 213–218
USS Dallas Los Angeles-class nuclear-powered attack submarine, 211, 212f
wxPython, 212
Port scan
basic categories, 225, 226t

“def portScan(event)”, 228–229, 228b execution with display all selected, 234, 234f
execution with display NOT selected, 234, 234f
GUI, 226, 228f
HTTP, 225
program launch, 226, 228f
registered ports, 226, 227t
“Setup the Application Windows”, 228–229
TCP/IP, 225
well-known ports, 226, 227t
PrintBuffer function
alpha string, 106
Hex and ASCII, 106
PSNMT. See Python silent network mapping tool (PSNMT)
Python Anywhere
Home Page, 290, 292f
Multi Core Execution Results, 298, 299f Single Core Execution Results, 298, 299f
Python-based solutions, 308
Python forensics
baTarget, 95, 95f
binary file/stream, 95
bytearrays, 95
capture.raw, 108
chLog.log, 107
code walk-through (see Code walk-through, Python forensics)
cost and barriers, 8–9
cybercrime investigation challenges, 2–6 Daubert evidence, 9–10
Daubert standard, 15
deductive/inductive reasoning, 92
digital crime scenes, 92–93
digital investigations, 1
disk image, 93–94
domain-specific researchers, 6
environment set up, 14–15
Execution test directory, p-search, 107, 108f forensic/digital investigation, 16–17
Hex/ASCII representation, 99–100
hypothesis, 93

IDE, 37–46
iMac, 108, 111f
indexing methods, 91–92
indexOfWords, 114
initialization code, Matrix, 113–114 installation, 17–24
interface software, 14
kryptonite, 94
len function, 95
library modules, 96
lifecycle positioning, 8
logger, 99
main function, 98
matrix file, 112–113
mobile devices, 46–50
narc.txt, 94
open source and platform independence, 8 operating system/supporting libraries, 14 organization, 10
ParseCommandLine, 98
PrintBuffer functions, 99
programming language popularity, 6, 7f p-search, 96–98, 97f, 98f
Python 2.x to 3.x versions, 16
requirements, program, 96, 96t
SearchWords function, 98–99
Shell, 7–8, 16
source code, 1–2
Standard Library, 25–26, 27–36
standard library mapping, 96, 97t
strings, 110
technology-based solutions, 6
test-then code-then validate, 6, 7f
third-party packages and modules, 36–37 Ubuntu Linux, 108, 110f
unicode, 16
virtual machine, 51
weighting approach, 110, 111f
Windows installation, 17–24
WingIDE environment, 99–100, 99f WordProbable(word), 112
Python forensics application
class names, 56
constants, 55

Cryptographic SmartCard, 54, 54f
CSVWriter (class), 63, 74–75
design considerations, 59–64, 60t
directory after pfish execution,
83–84, 84f
fixed-sized font, 64
Full code listing pfish.py, 75–76
Full code listing _pfish.py, 76–83
functions name, 55
fundamental requirements, 58
global variable name, 55
HashFile, 71–74
local variable name, 55
logger, 63
main function, 62
module, 55
object name, 55
“one-way file system hashing”, 56–64 parsecommandline, 62
ParseCommandLine( ), 66–69
pFishLog file., 86, 86f
program structure, 61–62
Standard Library Mapping, 59, 60t test run of pfish.py., 83, 84f
ValiditingDirectoryWritable, 69
WalkPath, 69–71
walkpath function, 63
writing the code, 63–64
Python Image Library (PIL)
download, Windows, 126, 127f
EXIF GPSTAGS, 133–137
EXIF TAGS, 131–133
Ubuntu 12.04 LTS, 128, 129f
Windows installation Wizard, 128, 128f
Python installation
customization user manual, 19, 21f directory snapshot, 22–23, 23f
download confirmation, 18, 18f
Graphical User Interface (GUI) module, 19 Hello World, 23–24, 24f
installation, Python 2.7.5., 22, 22f
programming language official web site, 17–24, 17f
Python 2.7.5 installer, 18, 19f

TCL/TK install, 19, 21f
user selection, 19, 20f
Windows downloading, 17, 17f
Windows taskbar, 23, 23f
Python language, 307–308
Python shell
built-in data structures, 16
definition, 31
hex values, 28f
iOS, 46–47, 47f
session, hex( ) and bin( ), 27f
Python silent network mapping tool (PSNMT) classLogging.py source code, 257–258 commandParser.py, 256–257
comma-separated value (CSV) file, 249 csvHandler.py source code, 258
data types, 249
decoder.py source code, 253–256
psnmt.py source code, 249–253
Standard Library signal module, 248 UDP packet header, 247, 248f
Wing-IDE environment, 249, 250f
The Python Standard Library
built-in constants, 31–32
built-in exceptions, 33–34
built-in functions, 27
built-in types, 32–33
cryptographic services, 35
data compression and archiving, 34 file and directory access, 34
file formats, 35
hash values, 25
OS services, 35–36
platform-specific APIs, 25
SHA256 hash value, 26, 26f
Ubuntu Linux execution, 26, 27f
Python Standard Libraries, 290
Python version 2.x, 30, 30t

R

Rainbow Table generation
Multicore password generator, 283–287 Plaintext Rainbow Table output, 287,

287f Single core password generator code, 280–283

Raw sockets, Python
built socket method, 244
human readable forms, 244
IPv4 packet header, 243, 243f
Linux, 241–242
promiscuous/monitor mode, 240–241 Standard Library documentation, 243
TCP/IP packet contents, 242, 242f
TCP packet header, 245, 246f
unpacking buffers, 242–247

Remote Switched Port ANalyzer, 237–238
S

SearchWords function
baTarget, 104
bytearray, 106
command line arguments, 103
forensic log file, 104
newWord variable, 105
PrintBuffer function, 106
p-search code, 103
size characteristics, 105

Single Core Rainbow, 295–296
Sockets
API, 206
client.py code, 209–210
description, 206
IP addresses, 206, 207
localhost loopback, 208, 208f
Python language capabilities, 207
Python programs, 208
server.py code, 208–209
simplest local area network, 206, 206f
SPAN. See Switched Port ANalyzer (SPAN)
Standalone Linux Single, 298, 299f
Standard Library modules, 292–295

Strftime Output Specification, 174t
Switched Port ANalyzer (SPAN), 237–238, 239f

T

Time module
attributes and methods, 169 epoch date, 169
strftime method, 173
UTC/GMT time, 170, 171 zone designation, 172

Twisted matrix (TWISTED), 37
U
Ubuntu Linux
download Web page 12.04 LTS,

44 , 44f
Linux-based tools, 42
software center, 44, 45f
terminal window, 44, 45f

Universal Time (UTC)
calculation, hours, 172
diplomatic treaty, 166
and GMT (see Greenwich Mean Time (GMT))

User Datagram Protocol (UDP)
forensic log, 261
output file, Excel, 261, 262f
packet header, 247, 248f

UTC. See Universal Time (UTC)
V
Validation/certification process, 308 ValiditingDirectoryWritable, 69
W

WalkPath, 69–71
Windows installation Wizard, 128, 128f Windows 8 phone, 49–50
WingIDE

in action, 41, 41f

auto complete feature, 42, 43f code, 41, 42f
completed list, 41, 43f
flavors, 39
project, 137, 140f
Python Shell display, 39, 40f
WingIDE 4.1 Personal, 39, 40f

for Jy

Graphical Jython UI and S
Java Swing and WebSphe

g
thon

Scripts Development using
ere Application Server

Robert A. Gibson

Apre

SS®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Contents at a Glance

About the Author
xix About the Technical Reviewers
xxi Introduction
xxiii

■ Chapter 1: Components and Containers
■Chapter 2: Interactive Sessions vs Scripts
■Chapter 3: Building a Simple Global Security Application
■Chapter 4: Button Up! Using Buttons and Labels
■Chapter 5: Picking a Layout Manager Can Be a Pane
■Chapter 6: Using Text Input Fields
■Chapter 7: Other Input Components
■Chapter 8: Selectable Input Components
■Chapter 9: Providing Choices, Making Lists
■Chapter 10: Menus and MenuItems

■Chapter 11: Using JTree to Show the Forest: Hierarchical Relationships of Components

■ Chapter 12: Motion to Take from the Table: Building Tables
■Chapter 13: Keystrokes, Actions, and Bindings, Oh My!
■Chapter 14: It’s the Event of the Year: Events in Swing Applications
■Chapter 15: Nuts to Soup: Using Jsoup to Enhance Applications
■Chapter 16: Conversing with a User with Dialog Boxes
■Chapter 17: Specialized Dialog Boxes
v

Contents at a GlanCe

■ Chapter 18: Monitoring and Indicating Progress
■Chapter 19: Internal Frames
■Chapter 20: Building a Graphical Help Application
■Chapter 21: A Security Configuration Report Application
■Chapter 22: WASports: A WebSphere Port Application

Index
vi

Introduction

A long time ago, in a galaxy far, far away...

Okay, maybe not so long ago, unless you are thinking in terms of “web years.” In 1995, Java was introduced into the world. At that time, it included a Graphical User Interface (GUI) framework called the Abstract Window Toolkit (AWT). Unfortunately, the AWT contained many native widgets that depended on the underlying operating system.

It didn't take long to realize that the AWT approach was limited, and unfortunately, unreliable. So, in 1997, the Java Foundation Classes (JFC) was introduced and included the Swing component classes.[1]
Late in 1997, Jython was created to combine the performance provided by the Java Virtual Machine (JVM) with the elegance of Python. In so doing, the entire collection of existing Java classes were added to the ones available from Python. What this means to a software developer is that the rapid prototyping and iteration of Python can be combined with the impressive class hierarchy of Java to provide an amazing software development platform.
In 2002, Samuele Pedroni and Noel Rappin wrote a book titled *Jython Essentials* which, interestingly enough, uses an example similar to the one shown in Listing 1.

Listing 1. Welcome to Jython Swing

```
wsadmin>from javax.swing import JFrame
wsadmin>win = JFrame( "Welcome to Jython Swing" ) wsadmin>win.size = (
400, 100 )
wsadmin>win.show()
```

The output of this interactive wsadmin code snippet is shown in Figure 1.

Figure 1. *Welcome to Jython Swing application output*

At the time of this writing, the first chapter of Jython Essentials was (still) available on the O'Reilly website.[2] So, it has been obvious, at least to some people, just how valuable and powerful Jython can be as a Swing development environment.

[1] I hadn't realized, at least until I looked at *Java Foundation Classes in a Nutshell* by David Flanagan, just how much of the JFC was composed of Swing classes and components. Most of Part I of that book documents the GUI Application Programming Interfaces (APIs) used in client-side Java programs.
[2]See http://oreilly.com/catalog/jythoness/chapter/ch01.html.

Why Read This Book?

You may ask, if the *Jython Essentials* book has been around for more than a decade and talks about using Swing with Jython, is this book really necessary? The answer to that question is yes, because *Jython Essentials*, as well as the other Jython books, talk a little about using Swing with Jython and provides occasional example programs, but this topic is mentioned only in passing.

There are some other books about Jython, most notably *The Definitive Guide to Jython: Python for the Java Platform* by Jim Baker, Josh Juneau, Frank Wierzbicki, Leo Soto, and Victor Ng (Apress, ISBN 978-1-4302-2527-0). It too has some examples of using Swing with Jython. Unfortunately for the person interested in learning how to write Jython Swing applications, the amount of information is limited.

What Does This Book Cover?

The focus of this book, on the other hand, is to show you how to use Swing to add a GUI to your Jython scripts, with an emphasis on the WebSphere Application Server wsadmin utility. In fact, we teach you Swing using Jython and do it in a way that will make your scripts easier for people to use, more robust, more understandable, and therefore easier to maintain.

The Swing hierarchy is a huge beast.
How do you eat an elephant? One bite at a time.
That's what you're going to do with Swing in this book—consume it in a lot of small bytes.
In order to make it more easily consumable, the book uses lots of examples, with most of them building on earlier

ones. In fact, by the time you're done, you'll have touched on the almost 300 scripts that were written during the creation of this book.

Additional challenges exist, for example, event handling and threads. These too require some clarifying examples and explanation. We will also be dealing with concurrency, especially in the context of using threads in the applications that are created.

As you progress, you'll see that there are often a number of different ways to do

things. We try to point some of these different approaches, depending on the context. Why only some? Well, unfortunately, we are rarely able to identify them all. As I found after reading lots of different programs, there is often yet another way to do something. So, we admit that we don't know everything. In fact, that's one of the things that we find neat and interesting about writing code. We love to learn and hope that you do too.

Swing development is a "target rich" environment.
These days, it's a challenge to find command-line only programs. Wherever you look, programs have a graphical interface and allow the users to use their mouse to make selections. Frequently, you can use the mouse to completely specify the information required by a program to perform the user-desired operations. How many times have you been able to use a mouse to make all of the selections from the displayed information? I bet you don't have to think too hard to come up with a number of examples of this kind of interaction.
Unfortunately, this has not been the case for most WebSphere Application Server (WSAS) administrative scripts. When using wsadmin to execute one of these WSAS administrative scripts, developers have been forced to do one of the following:

• Provide command-line options as input.
• Use some kind of input file (such as a properties file, Windows .ini file, and so on).
• Have the script prompt the users and wait for them to provide an appropriate response.

This book is going to help you change all that. We're going to cover all of the information that you need to help you add a Graphical User Interface (GUI) to your WSAS Jython scripts. Does that mean that we cover each and every Java Swing class, method, and constant? No, unfortunately not. Take a look at *Java Swing* by Robert Eckstein, Marc Loy, and Dave Wood (ISBN 1-56592-455-X); it's more than 1,000 pages long! And, it's not the only huge book on Java Swing. Unfortunately, this is part of the problem. Many people are intimidated by the amount of information and are unsure of how and where to start.

One thing that you should realize is that we don't have to create a huge tome about each and every aspect on this subject in order to make it useful. For one,

we don't duplicate information that is available elsewhere. What we do need to do is show you:

• What is possible
• What is required
• How to make use of existing information

• How to take Java examples and produce equivalent (possibly even better) Jython scripts that do the same kind of thing

And that is what we intend to do with this book. How does that sound?

What You Need

What is required?

This book is all about using the Java Swing classes in your Jython scripts. The fact that a number of examples use the IBM WebSphere Application Server (WSAS) to demonstrate different things does not mean that you must have WSAS in order to use Swing in your Jython scripts. I happen to use the WSAS environment to demonstrate some of the more complete applications. So it is important to note that some, but not all, of the scripts included with this work depend on information that is provided by a WSAS environment. If you are interested in using the information in this book in your own Jython scripts, I encourage you to do so.

All of the book's scripts have been tested using WSAS versions 8.5, 8.0, and 7.0. Some of them are also usable on version 6.1, but there are some things that don't exist in that version of wsadmin.[3] When these issues pop up, they are addressed.

Most contemporary software programs have a graphical user interface. In fact, some people (like my kids) would be stymied by something like a Windows or UNIX command prompt. They would likely ask something like, "What am I supposed to do now? There's nothing to click on!" That's what this book is all about—helping you create user-friendly Jython scripts using Java Swing classes.

[3]Most notably, any scripts that depend on the SwingWorker class won't work on

version 6.1 of WSAS since that class is not available in the 6.1wsadmin class hierarchy.

xxv **Chapter 1**

Components and Containers

Before you begin your exploration of Swing objects and classes, I need to first explain how I am going to describe these things. For the most part, the objects that you use on your graphical applications are called *components*. In some places, they may be referred to as widgets. I'll try to be consistent and stick with components in hope of minimizing confusion.

In this chapter, you get your first exposure to the Swing hierarchy and see how a Jython application can use the Swing classes to create a Graphical User Interface (GUI). You'll begin with top-level container types and see what makes them special. Then you will see how Jython can help you to investigate and understand the class hierarchy. Next, you create a trivial application using an interactive session. You also get your first exposure to some of the challenges associated with the positioning of components on an application when you aren't aware of class default values. Finally, you'll see how users can impact the way that things are displayed should they resize the application window.

Another thing that you need to realize is that the Java Swing classes are not a complete replacement of the AWT. There are a number of places, as you will see, where AWT features continue to be used. For example, the AWT eventhandling elements and mechanisms are an integral part of the user interface that most people consider "Java Swing." When AWT features are needed, they are identified accordingly.

As you will soon see, you will be building applications using Swing components placed in a way to convey information to the users. Sometimes the users can interact with these components in order to provide information to the application. At other times, the components are used only to provide information to the users. An example of this kind of component is text placed on the application near an input field to direct users as to what kind of information, or input, is expected.

Sometimes, multiple components or objects are grouped together and associated with one another. An example of this is a list of some sort that's used to make a

selection. This grouping of components will be associated with, and contained within, a collection. One of the many concepts explained in this book is how to tell the Swing classes how a collection should be displayed. In fact, in order for a component to be visible, it must be associated with a container of some sort. Because of this, a hierarchy of containers and components exists in every Swing application. At the top of the hierarchy there needs to be a root, or top-level, container that holds the complete collection of application components.[1] This might make a little more sense when you see some examples.

Top-Level Containers

Some Swing containers are special. The main difference between these and other containers in the Swing hierarchy is that none of the top-level containers are descended from the javax.swing.JComponent class. In fact, each of them is a descendant of an AWT class. Because of this, these containers may not be placed into any other container. All of these special containers are called “top-level” containers because they are at the top (relatively speaking) of the application hierarchy.

[1]The biggest difference between collections and components is that a collection is a kind of component that can hold other components.

What are these container classes, and what does it mean that they aren’t descended from javax.swing. JComponent?
Take a look at the Java class documentation (i.e., the output of the Javadoc tool that’s used to generate API documentation in HTML).[2] In this documentation, you need to locate the top-level containers. Table 1-1 shows the top portion of the class hierarchy for each of the top-level containers.

Table 1-1. *Top-Level Containers* **Container Type** JApplet[3]

JDialog[4]
JFrame[5]
JWindow[6]
Class Hierarchy

java.lang.Object java.awt.Component java.awt.Container java.awt.Panel
java.applet.Applet javax.swing.JApplet java.lang.Object java.awt.Component

java.awt.Container java.awt.Window
java.awt.Dialog
javax.swing.JDialog java.lang.Object java.awt.Component java.awt.Container
java.awt.Window
java.awt.Frame
javax.swing.JFrame java.lang.Object java.awt.Component java.awt.Container
java.awt.Window
javax.swing.JWindow

As you can see, each of these classes descends from a java.awt class, not from the javax.swing.JComponent class. As with all AWT classes, this means that there is a significant portion of the class that is composed of native (i.e., operating system-specific) code.

[2] See http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html.
[3]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JApplet.html.
[4]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JDialog.html.
[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html.
[6]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JWindow.html.

Looking at these top-level components might make you wonder about the difference between a window (JWindow) and a frame (JFrame). Listing 1-1 shows a trivial interactive wsadmin session[7] that can be used to display a JWindow instance of a specific size.[8]

Listing 1-1. Simple JWindow example

```
wsadmin>from java.awt import Dimension
wsadmin>from javax.swing import JWindow
wsadmin>win = JWindow()
wsadmin>win.setSize( Dimension( 400, 100 ) )
wsadmin>win.show()
```

Wait a minute. How did I know that I needed to import the Dimension class from the java.awt library and then instantiate one of them in order to invoke the setSize() method?
The answer is I cheated. I first tried to do it without importing the Dimension class, as in Listing 1-2.

Listing 1-2. The setSize() exception

```
wsadmin>from javax.swing import JWindow
wsadmin>win = JWindow()
wsadmin>win.setSize( ( 400, 100 ) )
WASX7015E: Exception running command: "win.setSize( ( 400, 100 ) )";
exception information: com.ibm.bsf.BSFException: exception from Jython: ...
setSize(): 1st arg can't be coerced to java.awt.Dimension
```

Did you notice how the exception tells you exactly what you need to use to resolve the issue? This demonstrates just how easy it is to use an interactive wsadmin (or Jython) session to develop and test your applications.[9]
Now, getting back to the JWindow. If you execute the steps shown in Listing 1-1, you'll notice how empty it is. It is a completely blank slate. This provides you with the opportunity to completely define how your application will look. The tradeoff though is that you have to define each and every aspect of the application. For this reason, however, I prefer to use the JFrame as a starting point (at least for now), since it does a lot of the work for me. In fact, the vast majority of scripts in the remainder of the book use the JFrame class.

Getting Help from Jython

You just looked at the Java class documentation for the top-level containers. Do you really have to use the documentation, or is there any way to get to this kind of information from Jython? Let’s take a quick look at what Jython can do to help you. Listing 1-3 shows an interactive wsadmin session that includes the definition of a simple Jython class function (called classes) that can display information about the class definition hierarchy. In this case, this function is used to show information about the JFrame class.

[7] To start an interactive session, execute either the wsadmin.bat or wsadmin.sh shell script and identify the scripting language to be used as Jython. For example, "./wsadmin.sh -conntype none -lang Jython".
[8]I did not include a figure of the result because it is simply a plain white rectangle displayed in the top-left corner of the screen. Go ahead and try it for yourself, and you’ll see what I mean.
[9]You'll find that we often use script and application interchangeably in this book.

Listing 1-3. The classes function

```
wsadmin>from javax.swing import JFrame wsadmin>
wsadmin>def classes( Class, pad = '' ) : wsadmin> print pad + str( Class )
wsadmin> for base in Class.__bases__ : wsadmin> classes( base, pad + '| ' )
wsadmin>
wsadmin>classes( JFrame )
javax.swing.JFrame
| java.awt.Frame
| | java.awt.Window
| | | java.awt.Container
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver | | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | javax.accessibility.Accessible | | java.awt.MenuContainer
| javax.swing.WindowConstants
| javax.accessibility.Accessible
| javax.swing.RootPaneContainer
wsadmin>
```

How does this information compare with what you saw earlier from the Java class documentation? One difference is that the information that is displayed is from the specified class, which in this case is JFrame, down toward its ancestor components. It is also more complete. The really neat thing about this is that you are using the power of Jython to understand what is available from the Java Swing hierarchy.

How and Why Are You Able to Do This?

Java includes some properties that allow objects and classes to be dynamically "queried" to determine information about what the objects and classes can do. These properties are called *reflection* and *introspection*. The classes function in Listing 1-3 illustrates one way to use these properties. I won't get into this too much, at least at this point, but you will be seeing more about how to use some of the properties later.[10]

What's Next?…Starting Simple

After starting with a frame, you need to decide what your application should display. Let's start by adding the ubiquitous "Hello World!" message. How do you go about doing that? First, you have to figure out what kind of object you need to display text.

Swing has a component for that, called a JLabel. (You will soon realize that many of the Java Swing components start with a capital J).
[10]One important point to mention is that the output produced by the classes function when Jython (not wsadmin) is used is much harder to read. I chose to use the wsadmin environment for simplicity and readability.

Before you create a JLabel object, you need to tell Jython where it can obtain details about this class, just like you did for the JFrame class. Again, you use a variation of the Jython import statement. Listing 1-4 shows one way to do exactly this.

Listing 1-4. Adding a JLabel to the application

```
wsadmin>from javax.swing import JFrame
wsadmin>from javax.swing import JLabel
wsadmin>
wsadmin>frame = JFrame( 'Hello world' )
wsadmin>label = frame.add( JLabel( 'Hello Swing world' ) )
wsadmin>frame.pack()
wsadmin>frame.setVisible( 1 )
wsadmin>
```

The steps used in this example can be described as follows:
1. Instantiate (create) a JFrame object (supply the title). This is done by calling the JFrame constructor.
2. Instantiate a JLabel object (supply the label text). This is done by calling the JLabel constructor.
3. Add the JLabel object to the JFrame (application) object. This is done by calling the JFrame add() method and passing the JLabel object to it.[11]
The result of executing the code in Listing 1-4 is a small application window located in the top-left corner of the screen. first image in Figure 1-1

Figure 1-1. *The*

“Hello Swing world” window

The shows what this window looks like. It’s interesting to note, however, that the application title, which is normally on the title bar, is obscured by the application icons. The second image in Figure 1-1 shows what happens when you grab the right side of the window and drag it to the right, thereby increasing the window width and making the application title visible.

Adding a Second Label

Hopefully, that seems pretty straightforward to you. Let’s add another label just to see what happens. Listing 1-5 shows a different interactive wsadmin session that does just that.
[11]The program saves the result of calling frame.add() into a variable simply to make the interactive session more readable. **Listing 1-5.** Adding a second JLabel to the application

```
wsadmin>from javax.swing import JFrame
wsadmin>from javax.swing import JLabel
wsadmin>
wsadmin>frame = JFrame( 'Hello world' )
wsadmin>label = frame.add( JLabel( 'Hello Swing world' ) ) wsadmin>label =
frame.add( JLabel( 'Testing, 1, 2, 3' ) ) wsadmin>frame.pack()
wsadmin>frame.setVisible( 1 )
wsadmin>
```

Figure 1-2 shows the output. Unfortunately, it probably doesn’t look like you expected it to. What happened? The simple answer is that you didn’t tell the frame where to add the second label, so it put both labels in the same place, and Swing can’t show both labels in the same place. This chapter isn’t the best place get into details about Layout Managers; you’ll learn about them in Chapter 5. You’ll do just enough to get by here.

Figure 1-2. *After adding the second JLabel*

Is there a something simple that you can do make this example work? Yes there

is! Listing 1-6 shows how you can change the default Layout Manager used by the JFrame objects.[12]

Listing 1-6. Changing the default JFrame Layout Manager

```
wsadmin>from java.awt import FlowLayout
wsadmin>
wsadmin>from javax.swing import JFrame
wsadmin>from javax.swing import JLabel
wsadmin>
wsadmin>frame = JFrame( 'Frame title' )
wsadmin>frame.setLayout( FlowLayout() )
wsadmin>label = frame.add( JLabel( 'Hello Swing world' ) )
wsadmin>label = frame.add( JLabel( 'Testing, 1, 2, 3' ) )
wsadmin>frame.pack()
wsadmin>frame.setVisible( 1 )
wsadmin>
```

Unfortunately, as you can see in the image on the left in Figure 1-3, the two labels are side by side on the same line. If you drag the corner of the window to narrow the application window a little bit, you'll get better results. The image on the right in Figure 1-3 shows how the two labels are separate and distinct.

[12]In case you are interested, BorderLayout is the default Layout Manager used by JFrame.

Figure 1-3. *Adjacent JLabel objects*

Summary

What has this chapter taught you? For one thing, it shows how easily you can create a trivial graphical application using Jython and Swing. However, these aren't really good examples because they were created using interactive wsadmin sessions. In the next chapter, you'll learn about the differences between interactive sessions and script files.

Chapter 2

Interactive Sessions vs. Scripts

It shouldn’t take long for you to realize that you don’t want to be using interactive wsadmin sessions for your applications. What does it take? Well, there are some things of which you must be aware.

This chapter begins with some additional interactive scripts to help illustrate some of the important differences between the interactive environment and what is needed for your Jython Swing scripts to be successful. Part of this process includes using the Java compiler to understand that some methods have been deprecated. Unfortunately Jython doesn't warn you about this when the scripts using those methods are executed. Finally, you’ll take a look at thread safety and the challenge that it presents to developers.

Running Your First Script from a File

Let’s start by putting the trivial script from the previous chapter into a text file and then execute it using wsadmin.[1] Listing 2-1 shows the contents of the Welcome.py script file.

Listing 2-1. The Welcome.py Script File

```
from javax.swing import JFrame
win = JFrame( 'Welcome to Jython Swing' ) win.size = ( 400, 100 )
win.show()
```

What happens when you execute this script using wsadmin?[2] Nothing, that’s what. The question is, why don’t you see anything? The simple answer is that the call to the show() method returns immediately, and the script exits. There isn’t time for the Swing framework to display the instantiated window. To verify this, you can add a statement that causes the script to wait. The easiest and simplest way to do this is to use the raw_input() function to display a message and then wait for user input. Listing 2-2 shows the contents of the modified Welcome1.py script file.

Listing 2-2. The Welcome1.py Script File

```
from javax.swing import JFrame
```

```
win = JFrame( 'Welcome to Jython Swing' )
win.size = ( 400, 100 )
win.show()
if 'AdminConfig' in dir() :

raw_input( '\nPress <Enter> to terminate the application: ' )
```

[1]For those using Jython and not wsadmin, you don't need to use raw_input() to pause the script. [2]The command should look something like wsadmin -conntype none -f Welcome.py.

That's better! When you execute this script, the wsadmin utility stays around long enough for the application window to be displayed. Even though the application isn't too exciting, it is interesting enough as a starting point.

The next question you should ask at this point is, "What happens when you use (click on) the application close icon in the top-right corner of the application window?" The application exits, right? No, it doesn't. Look at the interactive wsadmin command prompt window. It continues to show the "Press <Enter> to terminate the application:" message. If you press the Enter key at this point, wsadmin exits and the operating system command prompt is displayed.[3]

How do you fix this behavior? How do you get wsadmin to exit when you use the application's close icon? The simple answer is that you have to tell it to. If you take a moment to think about it, you'll realize that the close icon is part of the JFrame. If you take a look at the JFrame online documentation,[4] you'll find the following:

Unlike a Frame, a JFrame has some notion of how to respond when the user attempts to close the window. The default behavior is to simply hide the JFrame when the user closes the window. To change the default behavior, you invoke the setDefaultCloseOperation(int) method.

Listing 2-3 shows the revised script, Welcome2.py. The only change from the previous script is the addition of a call to the setDefaultCloseOperation() method. What happens when you run this script and click on the close icon? The application window is removed, and the wsadmin utility terminates.

Listing 2-3. The Welcome2.py Script File

```
from javax.swing import JFrame
win = JFrame( 'Welcome to Jython Swing' )
```

```
win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
win.size = ( 400, 100 )
win.show()
if 'AdminConfig' in dir() :

raw_input( '\nPress <Enter> to terminate the application: ' )
```

This is perfect, right? No, not really. There are a couple of things that aren't as they should be.

Depending "Too Much" on Limited Information

The first problem with the trivial script in Listing 2-3 isn't very obvious. In fact, to figure out this problem, you can either:

- Look closely at the JFrame documentation.
- Write and compile an equivalent Java application.

Listing 2-4 shows an equivalent Java application.

[3]For readers who use Jython and not wsadmin, the Java process will still be executing in the background when you execute the script. [4]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html.

Listing 2-4. Welcome3.java

```
import javax.swing.JFrame;

public class Welcome3 {
public static void main( String args[] ) {
JFrame win = new JFrame( "Welcome to Java Swing" );
win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ); win.setSize( 400, 100 );
win.show();

}
}
```

What happens when you compile this? The warning messages (notes) shown in Figure 2-1 are generated.

```
Note: Welcome3.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

Figure 2-1. *Welcome3 warning messages*

If you recompile Welcome3 using the specified option, you get a more detailed explanation of the problem, as shown in Figure 2-2.

```
warning: [deprecation] show() in java.awt.Window has been deprecated
```

Figure 2-2. *Welcome3 detailed deprecation message*

Looking at the documentation for this show() method,[5] you'll find that you should be using setVisible(boolean) instead.

This shows[6] you a challenge that you'll encounter when using Jython to call Java methods. The Java compiler informs the users when a method has been deprecated, but the Jython environment does not. I'm not suggesting that you should avoid using calls to Java methods in your Jython scripts. Far from it' I'm just letting you know that you should check to see if a method has been deprecated before using it in your scripts. This situation is most likely to occur, at least from my experience, if you obtain an existing Java Swing program and translate, or convert, it to the equivalent Jython without checking for this kind of issue.

[5]Seehttp://docs.oracle.com/javase/8/docs/api/java/awt/Window.html#show%28%

[6]Pun intended.

Swing Threads

Deprecated methods aren't the only kind of issue of which you need to be aware. One of the most significant differences between simple applications and ones where the developer needs to be able to interact with users and respond to events is related to threads of control (aka threads). Articles have been around for quite some time that discuss issues related to the fact that Swing developers should be aware of the fact that most Swing components are not thread-safe,[7] and techniques exists for performing long-running operations.[8]

One technique is to define the application in a separate class and then wrap the instantiation of this class in a Java Runnable class. The calling of this Runnable class is deferred until the Swing environment is ready for it. This slight delay is performed by the Swing Event Dispatch thread and is initiated by a call to the invokeLater() method call of the SwingUtilities, or EventQueue class. This concept can be hard to follow. Take a look at an example of this technique, as shown in Listing 2-5.

Listing 2-5. Template1.py

```
import java
import sys
from java.awt import EventQueue
```

```
4|from javax.swing import JFrame
5|class Template1 :
6| def __init__( self ) :
7| frame = JFrame( 'Title' )
8| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
9| frame.pack()
10| frame.setVisible( 1 )
11|class Runnable( java.lang.Runnable ) :
12| def __init__( self, fun ) :
13| self.runner = fun
14| def run( self ) :
15| self.runner()
16|if __name__ in [ '__main__', 'main' ] :
17| EventQueue.invokeLater( Runnable( Template1 ) )
18| if 'AdminConfig' in dir() :
19| raw_input( '\nPress <Enter> to terminate the application: ' ) 20|else :
21| print '\nError: This script should be executed, not imported.\n' 22| if
'JYTHON_JAR' in dir( sys ) :
23| print 'jython %s.py' % __name__
24| else :
25| print 'Usage: wsadmin -f %s.py' % __name__
26| sys.exit()
```

A detailed description for this code can be found in Table 2-1. [7]See http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html. [8]See http://java.sun.com/products/jfc/tsc/articles/threads/threads2.html.

Table 2-1. *Template1.py Details*

Lines Detailed Description

1-4 Statements used to add class names to the Jython namespace.

5-10 User-defined application class (i.e., Template1).

11-15 Wrapper class, descended from java.lang.Runnable, that saves a class reference in the constructor and delays the instantiation of the class until the run() method is called (on the Swing Event Dispatch thread).

16-22 This is the (apparent) script-entry point. This code determines if the script was executed or imported. If imported, an error message is displayed and the script exits. If the script was executed, a call to instantiate the user application

class is deferred until the Swing Event Dispatch thread is ready to do so.

A roughly equivalent approach is shown in Listing 2-6.
Listing 2-6. Template2.py

```
import java
import sys
from java.awt import EventQueue
from javax.swing import JFrame
class Template2( java.lang.Runnable ) :
 def run( self ) :
 frame = JFrame( 'Title' )
 frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
 frame.pack()
 frame.setVisible( 1 )
if __name__ in [ '__main__', 'main' ] :
 EventQueue.invokeLater( Template2() )
 if 'AdminConfig' in dir() :
 raw_input( '\nPress <Enter> to terminate the application: ' ) else :
 print '\nError: This script should be executed, not imported.\n' if
'JYTHON_JAR' in dir( sys ) :
 print 'jython %s.py' % __name__
 else :
 print 'Usage: wsadmin -f %s.py' % __name__
 sys.exit()
```

Looking closely at these two listings, you should notice the differences. Instead of defining a separate user application class that creates the Swing application components in the class constructor, this code places all of the Swing component-creation operations in the class run() method.

The example scripts provided and described in this book tend to use the second template since it requires a little less code. It also makes a bit more sense, at least to me. While writing the example scripts in this book I found it extremely easy to start with the Template script and add the code that was required to demonstrate the topic being discussed. One useful feature of this technique is the fact that it should help you focus on the important differences and hopefully spend less time with the script as a whole.

Summary

This chapter shows what happens when you start using script files instead of interactive wsadmin sessions for your applications. The biggest difference is that you have to add a way for the script file to wait for the users to interact with the application. To do so, the script files will often use something like the raw_input() function provided by Jython. Additionally, subsequent script files in this book include an application class that defines the Jython components, containers, and structures used by the applications demonstrating the use of Swing classes and constructs.

Chapter 3

Building a Simple Global Security Application

In my experience, learning a programming topic is much easier when good examples are included. Throughout this book, you’re going to go through the process of building simple Jython Swing applications from scratch. As you do so, you will be learning about some of the available Swing components and using them to make your application useful. In this chapter, you take a quick look at a simple graphical application that displays whether the WebSphere Global Security has been enabled or not. This will lead you to the topics of panes and layers, which are so very important to Swing applications.

Adding Text to the Application Using a JLabel

At this point, you haven’t learned how to build a non-trivial, fully functional application. However, you do have information eno ugh to get started. How so? Well, you’ll start simple by building a simple application that displays the status of the global security. For those who may be unfamiliar with it, global security is simply a setting that determines whether a username and password is required to administer the application server environment. Listing 3-1 shows a complete Jython script to do just this.

Listing 3-1. SecStatus.py

```
import java
from java.awt import EventQueue
from javax.swing import JFrame
```

```
4|from javax.swing import JLabel
5|class SecStatus( java.lang.Runnable ) :
6| def run( self ) :
7| frame = JFrame( 'Global Security' )
8| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
9| security = AdminConfig.list( 'Security' )
10| status = AdminConfig.showAttribute( security, 'enabled' ) 11| frame.add(
JLabel( 'Security enabled: ' + status ) ) 12| frame.pack()
13| frame.setVisible( 1 )
14|if __name__ in [ '__main__', 'main' ] :
15| EventQueue.invokeLater( SecStatus() )
16| raw_input( '\nPress <Enter> to terminate the application: ' ) 17|else :
18| print 'Error: This script should be executed, not imported.\n' 19| print 'Usage:
wsadmin -f %s.py' % __name__
```

The statements that make use of wsadmin scripting objects are shown in lines 9 and 10. This application doesn't require a graphical user interface. In fact, it can easily be performed using a single wsadmin command with a print statement to display the status. Listing 3-2 shows a simple wsadmin command line that can be used to display the same security status information.[1]

Listing 3-2. wsadmin Command that Displays the Global Security Status

```
wsadmin -conntype none -lang jython -c "print 'Security enabled: ' +
AdminConfig.showAttribute( AdminConfig.list( 'Security' ), 'enabled' )"
```

However, you will quickly see that scripts that interact with users are often good candidates for graphical user interface, and so that is what you will be learning to do.

No "Pane," No Gain

When your application creates one of the top-level containers (e.g., a JFrame), a number of things are created by that container to help it perform its role. One of the things created by the JFrame constructor is a RootPane (which happens to be of type JRootPane).[2]

What does this RootPane do? It is composed of, and is used to manage, the following panes, each of which will be described in the following sections:

- A glass pane
- A layered pane

• A content pane
• An optional menu bar

When You Live in a Glass House, Everything Is a Pane

The glass pane can be used to intercept events.[3] For most applications, it is a transparent pane that doesn't get in the way of your visible application components. However, there may be a time when you want to have something made visible on the glass pane that covers an existing component on the content pane. There is an interesting example in the *Java Swing Tutorial*, on the :How to Use Root Panes" page,[4] that demonstrates one possible use of the glass pane. A section about the glass pane can be found on that same page.[5]

At this point, I haven't discussed enough of using Swing with Jython for the novice user to understand this particular example. However, someone with some knowledge of Java Swing and Jython might be interested in how an equivalent Jython script would look. For those folks, I have included a translated version of the GlassPaneDemo.py script to read and play with. Figure 3-1 shows some sample images that can be created using this script. The complete script is available in the source code for this chapter.

[1] The contents of Listing 3-2 must be entered as a single line / statement.
[2]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JRootPane.html.
[3]Which have not yet been defined. For the time being, you can think of things like mouse clicks as events, the first discussion of which appears in Chapter 4.
[4]See http://docs.oracle.com/javase/tutorial/uiswing/components/rootpane.html.
[5]See http://docs.oracle.com/javase/tutorial/uiswing/components/rootpane.html#glasspa

Figure 3-1. *GlassPaneDemo.py sample images*

The Layered Look Can Also Be a Pane

I'm certain that you have used an application that includes components that involve different layers. One of the most common instances of layers occurs

when a pop-up menu is displayed. It is important that this menu isn't obscured by any of the existing components. The layered pane is used to great effect for component positioning. Right now, you don't have to worry about this pane. It is discussed in Chapter 19.

The Optional MenuBar

Almost every graphical application that you can think of has some kind of menu. The MenuBar is a collection of MenuItems that are displayed to allow users to make selections. The applications that you create are likely to use MenuBars. In fact, many of the complete scripts that are developed in the last few chapters of this book use MenuBars and MenuItems. Since this is an important topic, the book spends all of Chapter 10 on menu-related issues.

The Content Pane Will Contain Most of the Visible Items

All Swing applications have at least one top-level container. Each of these top-level containers is the top, or root, of the application containment hierarchy (i.e., all of the visible components that appear in the container). In this book, almost all of the example applications will have a JFrame as the top-level container.

Additionally, just about every application will place the components to be displayed on the JFrame content pane. So, one of the most common things that needs to be done with a JFrame instance is to obtain a reference to its content pane. This is typically done using syntax like that shown in Listing 3-3.

Listing 3-3. Getting a JFrame Content Pane Reference contentPane = frame.getContentPane()

What kind of object is returned by this call? The simple answer is that a java.awt.Container[6] is returned. What, exactly, is a java.awt.Container? It certainly is a lot easier to comprehend than it is to describe. In fact, I described Swing containers as components that can hold a group of zero or more components.

[6]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html#getConte

Looking at the documentation,[7] you can see that it has six dozen methods. Granted, seven of these have been deprecated and another five are protected,

which means that you really shouldn't be using them from your Jython scripts. Another seven have multiple overloaded variations (e.g., there are five different add() methods). These add() methods are shown in Table 3-1.

Table 3-1. *Component add() methods*

Modifier and Type	Method and Description
Component	add(Component comp) Appends the specified component to the end of this container.
	Component add(Component comp, int index) Adds the specified component to this container at the given position.
	void add(Component comp, Object constraints) Adds the specified component to the end of this container.
	void add(Component comp, Object constraints, int index) Adds the specified component to this container with the specified constraints at the specified index.
	Component add(String name, Component comp) Adds the specified component to this container.

I won't be digging too deeply into these various methods, but I will be making good use of a number of them. In fact, you have already seen some example uses of the add() method (see line 11 of Listing 3-1, for example).

Did you catch that? I snuck one in on you there. If you were watching closely, you may have wondered why Listing 3-1 doesn't include a call to the getContentPane() method, as shown in Listing 3-3. How does this work? Well, the Swing developers realized that some method calls are very common, so they have provided a convenience. Instead of writing a statement like you see in Listing 3-4, you are allowed to leave out the call to the getContentPane() method.

Listing 3-4. Calling getContentPane() to add() Something

```
frame.getContentPane().add( something )
```

Where is this documented? Right on the JFrame documentation,[8] where you'll find the paragraph shown in Figure 3-2.[9]

[7] See http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html.

[8]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html.
[9]Yes, convenience is misspelled in the documentation.

... As a conveniance add and its variants, remove and setLayout have been overridden to forward to the contentPane as necessary. This means you can write:
frame.add(child);
And the child will be added to the contentPane.
Figure 3-2. *JFrame conveniences from the JFrame documentation*

Can you still include the call to the getContentPane() method in your code? Certainly. You aren't forced to use this convenience. If this makes your code easier and more understandable, that is your choice. Don't be surprised when most of the sample Java Swing code found in the wild[10] uses this convenience.

Summary

In this chapter, you saw a simple Jython Swing application that displays the current WebSphere global security status as the text of a JLabel component. Even though it is trivial in nature, it demonstrates the structure of the applications to be found throughout the book. Additionally, you had your first exposure to the layering that exists in the JFrame class, which will be used in almost every example application found in this book.

[10]The "wild" here refers to the wild, wild west, also known as the World Wide Web (WWW). **Chapter 4**

Button Up! Using Buttons and Labels

Up to this point, the applications you built in this book have been kind of boring.[1] They really haven't done anything except display a message. But now it's time to change that. In this chapter, you're going to add a button to your application and then have it react when the user presses it. How does that sound?

To do this you'll first take a look at the JButton class hierarchy and see the Java way to create a button and add it to the application frame. Then you'll see a verbose way, in Jython, to do the same thing. Then you'll see a more concise way to put a button on the frame and have an event handler update another

component when the user presses the button. This technique will allow your applications to react when the users click one of the buttons.

JButton Class Hierarchy

Before you start making buttons, it seems appropriate to take a look at this class. In Chapter 1, you saw a function named classes that displayed the class hierarchy for a given type from the inside out. Listing 4-1 shows this simple function again.[2]

Listing 4-1. classes.py

```
def classes( Class, pad = '' ) : print pad + str( Class )
for base in Class.__bases__ :

classes( base, pad + '| ' )
```

Using this function, you can display the JButton class hierarchy. Listing 4-2 shows an interactive wsadmin session that does just that.[3]

Listing 4-2. JButton class hierarchy

```
wsadmin>from javax.swing import JButton wsadmin>
wsadmin>classes( JButton )
javax.swing.JButton
| javax.swing.AbstractButton
| | javax.swing.JComponent
| | | java.awt.Container
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver | | | | | java.awt.MenuContainer | | | | |
java.io.Serializable
| | | java.io.Serializable
| | java.awt.ItemSelectable
| | javax.swing.SwingConstants
| javax.accessibility.Accessible wsadmin>
```

[1] Except, of course, for the GlassPaneDemo.py script mentioned in Chapter 3, which was converted from the original Java. But it doesn't count since I haven't covered the concepts used in that code.

[2] Remember that the output shown in this book is from the wsadmin version of Jython since it's easier to understand.

[3]The command used to start this wsadmin session included the -profile classes.py parameters.

What are you going to do with this information? Well, the first thing that you need to do is to create, or instantiate, a button instance. Then you'll add it to the frame. To begin, take a quick look at a trivial Java application to do this. Listing 4-3 contains the relevant code snippet.[4]

Listing 4-3. The Java Code to Create a Button and Add It to a Frame

```
JFrame frame = new JFrame( "ButtonDemo" );
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ); JButton button =
new JButton( "Press me" );
frame.add( button );
frame.pack();
frame.setVisible( true );
```

What would this look like in Jython? Listing 4-4 shows the equivalent Jython code.

Listing 4-4. The Verbose Jython Way to Do the Same Thing

```
frame = JFrame( 'ButtonDemo_01' )
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ) button =
frame.add( JButton( 'Press me' ) )
button.addActionListener( ButtonPressed() )
frame.pack()
frame.setVisible( 1 )
```

They certainly do look similar, don't they? What if you use an interactive wsadmin session to execute these steps? Listing 4-5 shows what happens when you try to do this.

Listing 4-5. The wsadmin interactive session to create and add a button

```
wsadmin>from javax.swing import JFrame, JButton
wsadmin>frame = JFrame( 'ButtonDemo_01' )
wsadmin>button = JButton( 'Press me' )
wsadmin>frame.add( button )
javax.swing.JButton[,0,0,0x0,invalid,alignmentX=0.0,alignmentY=...
```

[4]The complete application can be found in the code\Chap_04\ButtonDemo_01.java file.

The figure ends when some output is generated by the frame.add() method call. What was returned by the add() method call? Listing 4-6 shows a slightly different interactive session. This time, the value returned by the add() method call is saved into a variable named result. Then you determine if the value returned by add() is the same one passed in. Why? Well, the JFrame class[5] inherits five different add() methods from the java.awt.Container. This Jython statement is calling the Container add() method[6] that returns the same value that was passed in.

Listing 4-6. The wsadmin interactive JButton session 2

```
wsadmin>from javax.swing import JFrame, JButton wsadmin>
wsadmin>frame = JFrame( 'ButtonDemo' ) wsadmin>button = JButton( 'Press
me' ) wsadmin>result = frame.add( button )
wsadmin>result == button
1
wsadmin>
```

You can use this fact to simplify your code. Listing 4-7 shows this simplification. Instead of instantiating the JButton instance and saving the returned value in a variable (e.g., button), you can combine the add() call with the JButton constructor call and save the value returned from the add() call.

Listing 4-7. The wsadmin interactive JButton session 3

```
wsadmin>from javax.swing import JFrame, JButton
wsadmin>
wsadmin>frame = JFrame( 'ButtonDemo' )
wsadmin>frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
wsadmin>button = frame.add( JButton( 'Press me' ) ) wsadmin>...
```

■**Note** this technique doesn’t work for every Container add() method call, since not every add method returns a value.

The application found in Button_01.py contains a button that says “Press me!” that can be pressed. Unfortunately, it doesn’t do anything yet, but you’ll get there

right after a short digression. At this point it is appropriate to address an important question: How does Swing know where to put things on the application?

The Layout of the Land

One of the many parts of a Container is something called the Layout Manager. I touched on this a little in Chapter 1, and I will be going into detail about it in Chapter 5. For the moment, you'll simply take a quick peek "under the covers," so to speak.

For the time being, you're going to use JFrame as the top-level container. One way that you can learn more about the JFrame class hierarchy is to use the Java documentation. Figure 4-1 shows this relationship, at least the portion covered by the Java documentation.

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html.
[6]See http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html#add%28java.av

java.lang.Object
java.awt.Component
java.awt.Container
java.awt.Window
java.awt.Frame
javax.swing.JFrame

Figure 4-1. *JFrame class hierarchy*

The root of this class hierarchy is an Object, which isn't too surprising. The class hierarchy for the JFrame Swing class can be seen in Listing 4-8. Note, however, that unlike the JFrame Javadoc class hierarchy shown in Figure 4-1, the output of the classes function starts at the JFrame class and works its way up the hierarchy. So this is kind of an inside-out view of the hierarchy. One of the things that I found interesting about this view is the presence of additional classes that don't show up in the Javadoc.

Listing 4-8. Another look at the JFrame class hierarchy

```
wsadmin>from javax.swing import JFrame wsadmin>
wsadmin>classes( JFrame )
```

```
javax.swing.JFrame
| java.awt.Frame
| | java.awt.Window
| | | java.awt.Container
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver | | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | javax.accessibility.Accessible
| | java.awt.MenuContainer
| javax.swing.WindowConstants
| javax.accessibility.Accessible
| javax.swing.RootPaneContainer
wsadmin>
```

This diagram is kind of nice, but it would be more useful if it also allowed you to see the methods and attributes that exist at each level of the hierarchy. Better yet, it would be really nice if you had a way to filter the list of methods and/or attributes based on some text. Listing 4-9 contains a function that includes these additional capabilities. **Listing 4-9.** classInfo.py

```
1|def classInfo( Class, meth = None, attr = None, pad = '' ) :
2| print pad + str( Class )
3| prefix = pad + ' '
4| if type( meth ) == type( '' ) :
5| comma, line = ", " + prefix
6| methods = [
7| n for n, v in vars( Class ).items()
8| if n.lower().find( meth.lower() ) > -1 and callable( v )
9| ]
10| methods.sort()
11| for m in methods :
12| if len( line + comma + m ) > 65 :
13| print line.replace( '|', '>' )
14| comma, line = ", " + prefix
15| line += comma + m
16| comma = ', '
17| if not line.endswith( ' ' ) :
18| print line.replace( '|', '>' )
```

```
19| if type( attr ) == type( '' ) :
20| comma, line = ", " + prefix
21| attribs = [
22| n for n, v in vars( Class ).items()
23| if n.lower().find( attr.lower() ) > -1 and not callable( v ) 24| ]
25| attribs.sort()
26| for a in attribs :
27| if len( line + comma + a ) > 65 :
28| print line.replace( '|', '*' )
29| comma, line = ", " + prefix
30| line += comma + a
31| comma = ', '
32| if not line.endswith( ' ' ) :
33| print line.replace( '|', '*' )
34| for b in Class.__bases__ :
35| classInfo( b, meth, attr, pad + '| ' )
```

How do you use it? Well, when you call the classInfo() function, you must specify at least one parameter, which is a class (e.g., JFrame). In addition, you can als provide one or two parameters to show the methods and/or attribute names at each level of the class hierarchy. I find it best to include the parameter keyword (i.e., meth or attr) to indicate which parameter is being provided.

For example, if you want to display all of the methods in the JFrame hierarchy that contain the case-insensitive string "layout", you could call this function a statement, as shown on line 3 in Figure 4-7. Method names that include the specified text (i.e., "layout") are indicated using the > character.

Listing 4-10. JFrame class hierarchy showing the “layout” methods.

```
1 wsadmin>from javax.swing import JFrame
2 wsadmin>
3 wsadmin>classInfo( JFrame, meth = 'layout' )
4 javax.swing.JFrame
5 | java.awt.Frame
6 | | java.awt.Window
7 | | | java.awt.Container
8 > > > getLayout, setLayout
9 | | | | java.awt.Component
10 > > > > doLayout
```

```
11 | | | | | java.lang.Object
12 | | | | | java.awt.image.ImageObserver
13 | | | | | java.awt.MenuContainer
14 | | | | | java.io.Serializable
15 | | | javax.accessibility.Accessible
16 | | java.awt.MenuContainer
17 | javax.swing.WindowConstants
18 | javax.accessibility.Accessible
19 | javax.swing.RootPaneContainer
```

I wasn't too surprised by the presence of getter and setter methods in the layout (i.e., getLayout and setLayout), but I didn't expect to see the doLayout method in the java.awt.Component class. Looking at the Javadoc for the java.awt.Component class, you'll find the information about the doLayout() method[7] shown in Figure 4-2.

public void doLayout()Causes this container to lay out its components. Most programs should not call this method directly, but should invoke the validate method instead.

Figure 4-2. *java.awt.Component.doLayout() description*

[7]See http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html#doLayout%28%

That's kind of neat, but I don't intend to be calling that method in any of my code. Let's stick with the recommendation and avoid using it.

Since you looked for the layout methods, it's wise to take a moment to see if any attributes exist that refer to "layout". Listing 4-11 shows how to specify the attr keyword argument for the classInfo() function. The generated output of the JFrame class hierarchy shows that only one attribute exists in this hierarchy that contains the string "layout". It occurs in the java.awt.Container class.

Listing 4-11. JFrame class hierarchy showing "layout" attributes

```
wsadmin>from javax.swing import JFrame
wsadmin>
wsadmin>classInfo( JFrame, attr = 'layout' ) javax.swing.JFrame
| java.awt.Frame
| | java.awt.Window
| | | java.awt.Container
```

```
* * * layout
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver
| | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | javax.accessibility.Accessible
| | java.awt.MenuContainer
| javax.swing.WindowConstants
| javax.accessibility.Accessible
| javax.swing.RootPaneContainer
wsadmin>
```

What does this mean? Well, if you look at the Javadoc for java.awt.Container,[8] you might expect to see a "field"[9] called layout. So why isn't one there? Well, you can thank Jython for that. It automatically recognizes a property for which a getter (i.e., getPropertyName or isPropertyName) and a setter (i.e., setPropertyName) method are defined. This allows you to use the PropertyName as an expression or in an assignment statement. You'll use this feature throughout the examples in this book. Let's take a quick look at the example interactive wsadmin session shown in Listing 4-12.

[8]See http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html. [9]Java terminology for what is called an "attribute" in Jython programs. **Listing 4-12.** Huh? What's goin' on, Lucy?

```
1 wsadmin>from java.awt import FlowLayout
2 wsadmin>from javax.swing import JFrame
3 wsadmin>
4 wsadmin>fLayout = FlowLayout()
5 wsadmin>frame = JFrame( 'Title' )
6 wsadmin>frame.layout
7 java.awt.BorderLayout[hgap=0,vgap=0]
8 wsadmin>frame.layout = fLayout
9 wsadmin>frame.layout
10 java.awt.BorderLayout[hgap=0,vgap=0]
11 wsadmin>frame.getLayout()
12 java.awt.BorderLayout[hgap=0,vgap=0]
13 wsadmin>frame.getContentPane().getLayout() 14
```

```
java.awt.FlowLayout[hgap=5,vgap=5,align=center] 15
wsadmin>frame.getContentPane().getLayout() == fLayout 16 1
17 wsadmin>
```

Wait a minute! What's going on here? This demonstrates one situation where you have to be mindful of what Jython is trying to do with the "automatic" attribute when it finds getter and setter methods. Table 4-1 describes each line of the wsadmin interactive session shown in Listing 4-12.

Table 4-1. *What's Goin' on Lucy, Explained*

Lines Detailed Description/Explanation

1-3 Import the AWT and Swing classes that you will be using.
4 Instantiate a FlowLayout Layout Manager object (fLayout).
5 Instantiate a JFrame object and provide a title.

6-7 Use the frame.layout attribute to display information about the default Layout Manager that was created by the JFrame constructor.
Note: It is a BorderLayout Manager object.

8 Use the frame.layout attribute to assign the kind of Layout Manager you want the frame to use.
9-10 Use the frame.layout attribute to verify the Layout Manager associated with the frame. **Note:** It is *still* a BorderLayout Manager object.

11-12 Use the frame.layout getter method to determine which Layout Manager is associated with the frame.
Note: It is *still* a BorderLayout Manager object.

13-14 Use the frame.getContentPane() method to obtain a reference to the frame ContentPane, then use its layout getter method to determine the Layout Manager associated with the Content Pane. **Note:** It is a FlowLayout Manager object.

15-16 Verify that the FlowLayout Manager object being used by the Content Pane is, in fact, the same object that was created in line 4.
Note: The 1 in line 16 means true.
The code you are seeing here is due to a special convenience provided by the JFrame class developers.

The sentence from the Javadoc is shown in Figure 4-3.
As a convenience add and its variants, remove and setLayout have been

overridden to forward to the contentPane as necessary.
Figure 4-3. *JFrame "convenience"*
I'm bringing this to your attention now so that you are less likely to be surprised when you stumble upon it while you are developing and maintaining your Jython Swing scripts.

Buttons! Labels! Action!

You've looked at how to create a button and add it to your application. The next thing that you need to figure out is how to make the button do something when it's clicked. When a button is clicked, something called an "event" occurs. A button click is one of many events that can occur in a Swing application. For the time being, I'm going to focus[10] on this specific kind of event. You will learn about other events as they occur in the book.

Listing 4-13 shows a trivial Java application that prints a message (using the System.out.println() method call) when the button is pressed.
Listing 4-13. ButtonDemo_02.java : Reacting to a Button Press

```
import java.awt.event.*;
import javax.swing.*;
public class ButtonDemo_02 {
 public static void main( String[] args ) {
 javax.swing.SwingUtilities.invokeLater( new Runnable() {
 public void run() {
 JFrame frame = new JFrame( "ButtonDemo" );
 frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 JButton button = new JButton( "Press me" ); button.addActionListener(
new ActionListener() { public void actionPerformed( ActionEvent ae ) { 
System.out.println( "button pressed" ); };
 } );
 frame.add( button );
 frame.pack();
 frame.setVisible( true );
 }
 });
 }
}
```

Listing 4-14 shows the first attempt at implementing this application in Jython.

One of the things that Java includes is the concept of anonymous classes. Lines 10–14 of Listing 4-13 show how an anonymous class can be used to define an ActionListener, which contains an actionPerformed() method to be invoked when the buttonpressed event occurs. Jython doesn't allow anonymous classes. Listing 4-14 shows (in lines 7–9) one way to define a ButtonPressed descendent of the ActionListener class. In line 15, an instance of this class is instantiated and added as an action listener to the button. Many (most?) programmers, especially those new to Java, find the use of anonymous inner classes cumbersome and hard to read and write.

[10]Gaining and losing focus are two other kinds of events that can occur.

Listing 4-14. ButtonDemo_01.py: Reacting to a Button Press

```
1|import java
2|import sys
3|from java.awt import EventQueue
4|from java.awt.event import ActionListener
5|from javax.swing import JButton
6|from javax.swing import JFrame
7|class ButtonPressed( ActionListener ) :
8| def actionPerformed( self, e ) :
9| print 'button pressed'
10|class ButtonDemo_01( java.lang.Runnable ) :
11| def run( self ) :
12| frame = JFrame( 'ButtonDemo_01' )
13| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ) 14| button =
frame.add( JButton( 'Press me' ) )
15| button.addActionListener( ButtonPressed() )
16| frame.pack()
17| frame.setVisible( 1 )
18|if __name__ in [ '__main__', 'main' ] :
19| EventQueue.invokeLater( ButtonDemo_01() )
20| if 'AdminConfig' in dir() :
21| raw_input( '\nPress <Enter> to terminate the application:\n' ) 22|else :
23| print '\nError: This script should be executed, not imported.\n' 24| which = [
'wsadmin -f', 'jython' ][ 'JYTHON_JAR' in dir( sys ) ] 25| print 'Usage: %s
%s.py' % ( which, __name__ )
26| sys.exit()
```

Another approach is shown in Listing 4-15. This example, instead of defining a separate class, uses multiple inheritance. The class defined in line 8 is a descendent of the java.lang.Runnable and the java.awt. event.ActionListener classes. This means that it includes the run() method from the Runnable class and the actionPerformed() method from the ActionListener class. So all you need to do is identify the current class instance as the ActionListener, as shown in line 13.

Listing 4-15. ButtonDemo_02.py: Using Multiple Inheritance

```
1|import java
2|import sys
3|from java.awt import EventQueue
4|from java.awt.event import ActionListener
5|from java.lang import Runnable
6|from javax.swing import JButton
7|from javax.swing import JFrame 8|class ButtonDemo_02( Runnable,
ActionListener ) :
9| def run( self ) :
10| frame = JFrame( 'ButtonDemo_02' )
11| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ) 12| button =
frame.add( JButton( 'Press me' ) )
13| button.addActionListener( self )
14| frame.pack()
15| frame.setVisible( 1 )
16| def actionPerformed( self, e ) :
17| print 'button pressed'
18|if __name__ in [ '__main__', 'main' ] :
19| EventQueue.invokeLater( ButtonDemo_02() )
20| if 'AdminConfig' in dir() :
21| raw_input( '\nPress <Enter> to terminate the application:\n' ) 22|else :
23| print '\nError: This script should be executed, not imported.\n' 24| which = [
'wsadmin -f', 'jython' ][ 'JYTHON_JAR' in dir( sys ) ] 25| print 'Usage: %s
%s.py' % ( which, __name__ )
26| sys.exit()
```

Although this is pretty interesting, it’s not the only way that this feat can be accomplished. Next, you’ll take advantage of another Jython feature called “keyword arguments.”

Listing 4-16 shows that you don’t even have to declare the class as a descendent of the ActionListener class in order to use it as one (see line 7). You can simply use the implied setter (in line 12) to identify another class method as the ActionListener event handler. The really neat part of this, at least to me, is that you don’t have to call the method actionPerformed(); you can call it something more intuitive, like buttonPressed().

Listing 4-16. ButtonDemo_03.py: Multiple Inheritance Isn’t Required

```
1|import java
2|import sys
3|from java.awt import EventQueue
4|from java.lang import Runnable
5|from javax.swing import JButton
6|from javax.swing import JFrame
7|class ButtonDemo_03( Runnable ) :
8| def run( self ) :
9| frame = JFrame( 'ButtonDemo_03' )
10| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ) 11| button =
frame.add( JButton( 'Press me' ) )
12| button.actionPerformed = self.buttonPressed
13| frame.pack()
14| frame.setVisible( 1 )
15| def buttonPressed( self, e ) :
16| print 'button pressed'
17|if __name__ in [ '__main__', 'main' ] :
18| EventQueue.invokeLater( ButtonDemo_03() )
19| if 'AdminConfig' in dir() :
20| raw_input( '\nPress <Enter> to terminate the application:\n' ) 21|else :
22| print '\nError: This script should be executed, not imported.\n' 23| which = [
'wsadmin -f', 'jython' ][ 'JYTHON_JAR' in dir( sys ) ] 24| print 'Usage: %s
%s.py' % ( which, __name__ )
25| sys.exit()
```

The biggest difference between using the automatic, or implied, setter call in an assignment statement (e.g., line 12 in Listing 4-16) and using a call to the addActionListener() method (e.g., line 13 in Listing 4-15) is that using the call to the addActionListener() method is necessary when you need to register multiple ActionListener objects.

While you’re looking at some Jython techniques for creating a button, adding it to the frame, and adding an ActionListener, let’s take a look at some other things that you can do using Jython.
Listing 4-17 contains the run() method from two versions of the ButtonDemo_03 script. Lines 1–7 are from the ButtonDemo_03.py script file and lines 9-21 are from the ButtonDemo_03a.py script file.

Listing 4-17. Another Way to Create a Frame and Add a Button

```
1| def run( self ) :
2| frame = JFrame( 'ButtonDemo_03' )
3| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
4| button = frame.add( JButton( 'Press me' ) )
5| button.actionPerformed = self.buttonPressed
6| frame.pack()
7| frame.setVisible( 1 )
8|...
9| def run( self ) :
10| frame = JFrame(
11| 'ButtonDemo_03a',
12| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 13| )
14| frame.add(
15| JButton(
16| 'Press me',
17| actionPerformed = self.buttonPressed 18| )
19| )
20| frame.pack()
21| frame.setVisible( 1 )
```

In lines 2 and 3 (of Listing 4-17), you can see the technique that has been used up to this point to: Create a JFrame instance.•
• Provide a title string.
• Set the default close operation to be used when the close icon is selected.

This code corresponds with the commonly used Java style. Lines 10–13, on the other hand, show that you can do these same things with a single Jython statement. Granted, since I don’t want the lines to be too long, this statement is being displayed across multiple lines. However, it is still a single statement.
Lines 4 and 5 (again of Listing 4-17) show how you can:

• Create a JButton instance.
• Provide a text string to be displayed on the button.
• Add the button to the frame and save a reference to the button in a variable.
• Define the actionPerformed() method to be called when a button press event occurs.

The code in lines 14–19 does many of these same steps in a single statement. Again, multiple lines are used to make the various parts of the statement easier to read. The only significant thing that this statement doesn't do is save a reference to the button in a variable. This is simply because I didn't see the need to save that information. If you do need to save this value, you can have the code save the result of calling the frame.add() method in a variable.

Updating the Application

It seems kind of silly to have a GUI application display a message to the command prompt window when a button is pressed. This section explains what you need to do in order for the application to make changes to the user display when the button is pressed.

• Add a label to the application, the text of which can be modified by the event handler (the ActionListener actionPerformed()) method.
• Have the actionPerformed (i.e., buttonPressed()) method change the text in the label.

That sounds pretty straightforward, but there's something tricky you need to know about. In order for the buttonPressed() method to access the application label, it needs to be able to identify it and refer to it. So when you create the label, you need to save a reference to it in an instance attribute. Listing 4-18 contains the run() and buttonPressed() methods from the ButtonDemo_04.py application that does these things.

Listing 4-18. Parts of ButtonDemo_04.py

```
11| def run( self ) :
12| frame = JFrame( 'ButtonDemo_04' )
13| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
14| button = frame.add( JButton( 'Press me' ) )
15| button.actionPerformed = self.buttonPressed
```

```
16| self.label = JLabel( 'button press pending' )
17| frame.add( self.label, BorderLayout.SOUTH )
18| frame.pack()
19| frame.setVisible( 1 )
20| def buttonPressed( self, e ) :
21| self.label.setText( 'button pressed' )
```

Table 4-2 explains, in detail, each of the statements in Listing 4-19. The most important ones are in line 16, where a reference is saved to the label instance, and in line 21, where this reference is used to update the label text field.

Table 4-2. *ButtonDemo_04.py Explained*
Lines Detailed Description/Explanation

12-13 These are the same steps used previously to create a JFrame instance and to define the action to take when the close icon is selected.
14-15 These are the same steps used previously to create a button with some text and to identify the method to call when the button press event occurs.
16 Create a JLabel instance and save the reference as an object attribute value (i.e., self.label). 17 Add the JLabel instance to the frame, using the Layout Manager constant (i.e., BorderLayout.SOUTH),[11] to identify where on the application the label should be placed.
18-19 These are the same steps used previously to adjust the application size and to make the application visible.

20-21 ActionListener actionPerformed method that is invoked when a button pressed event occurs. Note how the object instance variable (self.label) is used to modify the text being displayed in the label field.

What happens when you execute this application? Figure 4-4 shows the application, before (on left) and after (on right) the e. This demonstrates that the technique performs as expected.

Figure 4-4. *ButtonDemo_04.py output images.*

Summary

This chapter demonstrates that you can choose to write Jython scripts using syntax that closely matches Java Swing constructs or you can use Jython idioms to make the applications more readable. This means that you should be able to use sample Java Swing applications or code snippets from the Internet to learn how to write and test your Jython Swing applications.

Additionally, you have started looking at some of the many Swing components and learning about how the Swing class hierarchy handles events such as button clicks. Chapter 5 delves into the topic of Layout Managers in great detail; they give you more control over how things are positioned on your application frame.

[11]http://docs.oracle.com/javase/8/docs/api/java/awt/BorderLayout.html. **Chapter 5**

Picking a Layout Manager Can Be a Pane

Earlier in this book,[1] you learned a bit about how items are placed, or positioned, on an application. This chapter deals with that topic in more detail and discusses some of the most commonly used Layout Managers. These include some of the more complicated ones, like the Absolute Layout and GridBagLayout Managers, as well as some of the simpler ones like the FlowLayout, BorderLayout, and BoxLayout Managers.

You will also learn about components that aren't exactly Layout Managers, including SplitPane and TabbedPane. You will then be better able to decide how to position the components on the application window in order to best communicate and interact with your users.

The Absolute Layout Manager Does Not Corrupt Absolutely

Designing and creating graphical applications can be frustrating. This is especially true when you have to position every component on the frame that is displayed to the users. Let's take a look at what this entails.

First, you need to remove any Layout Manager that's already on the container. Then, for each child component, you need to define its size and location on the

frame content pane. Finally, you need to call the container repaint() method to render the components within the container using the current details known about the objects.

This isn't particularly difficult, but it can be tedious and requires a lot of code. Listing 5-1 shows a portion of one way to do this.

Listing 5-1. AbsoluteLayout Example

```
6|class AbsoluteLayout( java.lang.Runnable ) :
7| def run( self ) :
8| frame = JFrame(
9| 'AbsoluteLayout',
10| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 11| )
12| frame.setLayout( None )
13| data = [
14| [ 'A', 20, 10, 0, 0 ],
15| [ 'B', 40, 40, 10, 10 ],
16| [ 'C', 80, 20, 20, 20 ]
17| ]
18| insets = frame.getInsets()
19| for item in data :
20| button = frame.add( JButton( item[ 0 ] ) ) 21| size = button.getPreferredSize()
22| button.setBounds(
23| insets.left + item[ 1 ],
24| insets.top + item[ 2 ],
25| size.width + item[ 3 ],
26| size.height + item[ 4 ]
27| )
28| frame.setSize(
29| 300 + insets.left + insets.right, # frame width 30| 150 + insets.top +
insets.bottom # frame height 31| )
32| frame.setVisible( 1 )
```

[1]Chapter 1 touched on the subject when you added a second label to the frame, In Chapter 4, you saw the layout-related methods and attributes.

How does it work? Well, line 12 shows how to remove any existing Layout Manager. Lines 19-27 show one relatively simple way to create some buttons and define their location and size on the containing frame. It's important to

remember, though, that the call to frame.add() is, in fact, equivalent to calling frame.getContentPane().add(), as discussed in Chapter 3.

When a component (e.g., a button) is created, its constructor will determine its "preferred" size. This preferred size can be obtained by calling the getPreferredSize() method for the component, as you can see in line 21. Then the component's setBounds() method can be used to locate and possibly resize the component.

The first two parameters of the setBounds() method define the component's position by identifying its top-left point. The next two parameters of the method identify the width and height of the component.
What happens when you run this script? Well, you will see three buttons positioned on the application frame. Figure 5-1 shows how the application frame should look. It also shows how challenging it can be to "do it yourself." Did you anticipate that some of the buttons would be overlaid on the screen? I certainly didn't.

Figure 5-1. *AbsoluteLayout example output*

Are there any advantages to using this technique? Yes, there are. If you resize the application, you will notice that the component's location and size don't change. This is not the case with every Layout Manager, as you will see.

Going with the Flow: The FlowLayout Manager

The next Layout Manager that you're going to investigate is quite simple to use and is called the FlowLayout Layout Manager.[2] This manager attempts to place the components in a row. If there isn't sufficient space in the container to do so, multiple rows will be used. If more than enough space exists, the Layout Manager will, by default, center the components horizontally in the available

space.

Listing 5-2 shows part of the FlowLayoutDemo.py sample application, which demonstrates how this works in an application.

Listing 5-2. FlowLayout Example

```
9|def run( self ) :
10| frame = JFrame(
11| 'FlowLayout',
12| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 13| )
14| cp = frame.getContentPane()
15| #-------------------------------------------------------16| # The alignment can be
one of the following values: 17| #-------------------------------------------------------
-18|# cp.setLayout( FlowLayout( FlowLayout.LEFT ) )
19|# cp.setLayout( FlowLayout( FlowLayout.RIGHT ) ) 20| cp.setLayout(
FlowLayout() ) # FlowLayout.CENTER = default 21|# cp.setLayout(
FlowLayout( FlowLayout.LEADING ) ) 22|# cp.setLayout( FlowLayout(
FlowLayout.TRAILING ) ) 23| for name in '1,two,Now is the time...'.split( ',' ) :
24| frame.add( JButton( name ) )
25| #-------------------------------------------------------26| # The
ComponentOrientation can be either LEFT_TO_RIGHT, or 27| #
RIGHT_TO_LEFT. The default is based upon system locale 28| #------------------
-------------------------------------29|# cp.setComponentOrientation( ... )
30| frame.setSize( 350, 100 )
31| frame.setVisible( 1 )
```

Here are some things to note about this code include:

• The FlowLayout class is part of the java.awt hierarchy.

• A default FlowLayout instance centers the components, with a small space between each.

• The component orientation (left-to-right vs. right-to-left) is inherited from the java.awt.Component class.

Figure 5-2 contains some sample outputs generated using this script.

[2] See http://docs.oracle.com/javase/8/docs/api/java/awt/FlowLayout.html.

Using default values.

Align = RIGHT, Orientation = Right to Left.

Align = RIGHT, Orientation = Right to Left

Too narrow to display the buttons on a single row.

Figure 5-2. *FlowLayout examples*

The description to the left of each image identifies the alignment and orientation values used to produce it. The FlowLayout Layout Manager is much easier than having to specify exactly where components should be placed. However, you must be mindful of what things will look like should the application change size.

South of the Border: The BorderLayout Manager

Next, take a look at the BorderLayout Layout Manager. What do layouts using this Layout Manager look like? Figure 5-3 shows a simple application using this manager.

Figure

5-3. *BorderLayout examples*

The BorderLayout Layout Manager enables you to easily position the components using some simple constants. The labels for each of the buttons shown in Figure 5-3 contain the names of these BorderLayout constants. Additionally, Listing 5-3 shows a portion of the script that generated this output.
Listing 5-3. BorderLayout Example

9| def run(self) :
10| frame = JFrame(
11| 'BorderLayout',
12| layout = BorderLayout(),
13| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 14|)
15| data = [
16| ['PAGE_START', BorderLayout.PAGE_START], 17| ['PAGE_END' , BorderLayout.PAGE_END], 18| ['LINE_START', BorderLayout.LINE_START], 19| ['LINE_END' , BorderLayout.LINE_END], 20|]
21| for name, pos in data :
22| frame.add(JButton(name), pos)
23| big = JButton(
24| 'CENTER',
25| preferredSize = Dimension(256, 128) 26|)
27| frame.add(big, BorderLayout.CENTER)
28| frame.pack()
29| frame.setVisible(1)

Tip i encourage you to execute the script and resize the application frame to see what happens to the buttons. remember that the results that you are seeing are because you're using the BorderLayout Layout Manager.

The BorderLayout Layout Manager has advantages, (e.g., simplicity of use) and disadvantages (e.g., the results of resizing the application may not be to your liking). Take these issues into account when you choose your application Layout Manager.

In addition to the constants shown on lines 16-27 in Listing 5-3, BorderLayout also includes some common directional constants. Figure 5-4 shows where these constants position components.

Figure 5-4. *BorderLayout directional constants*

Listing 5-4 shows a portion of the BorderLayoutNEWS.py script that was used to generate this output.

Listing 5-4. BorderLayoutNEWS Example

```
9| def run( self ) :
10| frame = JFrame(
11| 'BorderLayoutNEWS',
12| layout = BorderLayout(),
13| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 14| )
15| data = [
16| BorderLayout.NORTH,
17| BorderLayout.SOUTH,
18| BorderLayout.EAST,
19| BorderLayout.WEST
20| ]
21| for pos in data :
22| frame.add( JButton( pos ), pos )
23| big = JButton(
24| 'Center',
25| preferredSize = Dimension( 256, 128 ) 26| )
27| frame.add( big, BorderLayout.CENTER )
28| frame.pack()
29| frame.setVisible( 1 )
```

The only other point worth noting about the BorderLayout class is that you can

have the Layout Manager automatically separate components by specifying horizontal and vertical gap values when the Layout Manager is instantiated. You can see what this means by looking at Figure 5-5.

Figure 5-5. *BorderLayout with component separation*

What change was required? All you have to do is specify the gap values on the call to the BorderLayout constructor. The important line from the BorderLayoutGap.py script is shown in Listing 5-5. **Listing 5-5.** BorderLayoutGap Example 12| layout = BorderLayout(16, 8), The constructor used in Listing 5-5[3] identifies the horizontal and vertical gaps to be used between components.

What's in the Cards? Using the CardLayout Manager

The next Layout Manager you'll learn about is the CardLayout Manager. As with the previously described Layout Managers, this one lets you position the components. The interesting thing about this one is that like a deck of cards in a stack, only the top one is visible. What does this mean? Figure 5-6 shows a simple application using the CardLayout Manager. It uses the top row of buttons to allow users to select one of the cards to be displayed.

Figure 5-6. *CardLayout examples*

[3]See http://docs.oracle.com/javase/8/docs/api/java/awt/BorderLayout.html#BorderLay

Before you learn about the CardLayout Manager, I first have to quickly mention another container, specifically a JPanel[4] that is used in a number of places in this CardLayout application.

What is a JPanel? It is best to think of a JPanel as a simple, lightweight container

that can be used to hold a group of components. In this example, you will simply create JPanel instances and add some components to each.
Alright, let’s dig into the example. If you click on any of the buttons in the top row, you’ll see that the top panel doesn’t change but the bottom one displays the other components.
How do you do this? Note that the application, (i.e., the JFrame instance) uses one Layout Manager (on its ContentPane). In this case, the application will have two separate parts, or panes. Each pane will be a JPanel instance. The top is used to display the fixed buttons and the bottom is used to display the current CardLayout view.
Listing 5-6 shows the run() method for this example. In it, you can see how:

• The frame layout is configured to use a BorderLayout Manager instance.
• A reference to the frame ContentPane is obtained and the two support routines are called to populate the different parts of the frame panel.
Listing 5-6. CardLayoutDemo run() Method

```
class CardLayoutDemo( java.lang.Runnable ) :
 def run( self ) :
 frame = JFrame(
 'CardLayout',
 layout = BorderLayout(),
 defaultCloseOperation = JFrame.EXIT_ON_CLOSE
 )
 cp = frame.getContentPane()
 self.addButtons( cp, BorderLayout.NORTH )
 self.addCards( cp, BorderLayout.CENTER )
 frame.setSize( 300, 125 )
 frame.setVisible( 1 )
```

Listing 5-7 shows how the addButtons() method uses the default JPanel Layout Manager, i.e., a FlowLayout Manager instance, to add three buttons to the panel. The choice of button labels will become clear shortly. Please note that the ActionListener actionPerformed method for each button is assigned to be the buttonPress() method in this object instance. It is important to realize that each container component in the application has its own, possibly default, Layout Manager. This allows the application designers to position the individual components as they see fit.

Listing 5-7. CardLayoutDemo addButtons() Method

```
22| def addButtons( self, container, position ) :
23| panel = JPanel()
24| for name in '1,2,3'.split( ',' ) :
25| panel.add(
26| JButton(
27| name,
28| actionPerformed = self.buttonPress
29| )
30| )
31| container.add( panel, position )
```

Finally, the addButtons() method adds the button panel to the user-specified container, i.e., the frame ContentPane, using the specified location or position. [4]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/JPanel.html.

Listing 5-8 shows how the addCards() method uses a default JPanel instance for each of the card panels. For simplicity's sake, the first and last of these card panels contain only a label, which can be used to verify which card panel is being displayed. The middle card panel contains three more buttons to show how easily a card panel can be populated with components.

Listing 5-8. CardLayoutDemo addCards() Method

```
32| def addCards( self, container, position ) :
33| card1 = JPanel()
34| card1.add(
35| JLabel(
36| 'The quick brown fox jumped over the lazy dog.'
37| )
38| )
39| card2 = JPanel()
40| for name in 'A,B,C'.split( ',' ) :
41| card2.add( JButton( name ) )
42| card3 = JPanel()
43| card3.add(
44| JLabel(
45| 'Now is the time for all good men to come to...'
```

```
46| )
47| )
48| cards = self.cards = JPanel( CardLayout() )
49| cards.add( card1, '1' )
50| cards.add( card2, '2' )
51| cards.add( card3, '3' )
52| container.add( cards, position )
```

Finally, another JPanel is created to hold the various card panels that you just created and populated. Each of these card panels is added to the panel that is using the CardLayout Layout Manager. Note that when a panel is added, you specify an identifier to be used to select this panel. These strings correspond to the button labels used in the addButtons() method shown in Listing 5-7.

Listing 5-9 shows how the buttonPress() event-handling method can obtain a reference to the Layout Manager object instance being used by the panel. Then this Layout Manager's show() method can be used to specify a card panel instance to be displayed. The label of the button that was pressed can be used by calling the event.getActionCommand() method. All in all, this makes for a very clear, concise, and precise event handler.

Listing 5-9. CardLayoutDemo buttonPress() Method

```
53| def buttonPress( self, event ) :
54| deck = self.cards.getLayout()
55| deck.show( self.cards, event.getActionCommand() )
```

Hopefully you can see how easily this can be used to create a really neat application. Next, you will take a look at some of the various components that your applications can use.

Splitting Up Is Easy to Do: Using Split Panes

Unlike splitting the atom, Swing provides for a very easy way to partition all of, or part of, your application into two pieces. Unfortunately, it's not a Layout Manager, but it certainly applies to the topic at hand, so I cover it here anyway. The JPanel that was mentioned earlier provides you with a lightweight container for keeping a collection of

components together. Wouldn't it be neat if you had a simple way of splitting a panel in two, either vertically or horizontally? Well, there is a way, called a JSplitPane.[5] Figure 5-7 shows a very simple application with two buttons, one in each of the two parts of the horizontally split panel.

Figure 5-7. *JSplitPane sample output*

One feature that might not be immediately obvious in Figure 5-7 is that the separator (the divider) between the two parts of the split pane is moveable. The image on the right shows how the application appears after moving the separator as far to the right as possible. The limitation is imposed by the minimum size of each button.

Listing 5-10 shows just how easy it is to create and use one of these split panes in your applications.

Listing 5-10. SplitPane1: Simple Horizontal Separation

```
7|class SplitPane1( java.lang.Runnable ) :
8| def run( self ) :
9| frame = JFrame(
10| 'SplitPane1',
11| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
12| )
13| frame.add( JSplitPane(
14| JSplitPane.HORIZONTAL_SPLIT,
15| JButton( 'Left' ),
16| JButton( 'Right' )
17| )
18| )
19| frame.pack()
20| frame.setVisible( 1 )
```

Vertical Splits: Not as Painful as They Sound

It is just as easy to have a portion of your application window separated vertically. Look at line 14 in Listing 5-10. You can see how the JSplitPane class

contains a constant that indicates in which direction (horizontally or vertically) the pane should be split. If, instead of using JSplitPane.HORIZONTAL_SPLIT, you use JSplitPane.VERTICAL_SPLIT, your application will look similar to Figure 5-8.

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JSplitPane.html.

Figure 5-8. *JSplitPane2, vertical split*

Unlike in the previous application, this separator doesn't move. Well, that's only partially true. If you resize the window, as shown in Figure 5-9, you can see how the top part remains the same and all of the additional height is given to the bottom part. Then, you can move the separator bar down, resizing the two parts of the split panel. Again, each portion of the split pane is limited by the minimum size restriction of the components.

Figure 5-9. *JSplitPane2, vertical split and*

resizing

Limited Resources: Setting Size Attributes

A long time ago, in a galaxy far, far away, I took an economics class that used something called the “Guns versus butter model”[6] to describe the concept of choosing between two limited resources. In the same way, split panes restrict the amount of space that the separator bar can move. How is this determined? Every Swing component has the JComponent[7] and the Component[8] as some of its base classes. Therefore, each Swing component has different size attributes, as you can see in Listing 5-11. These attributes are used to identify things like the minimum, maximum, and preferred size of a component.

Listing 5-11. Component size attributes

```
wsadmin>from javax.swing import JButton
wsadmin>
wsadmin>classInfo( JButton, attr = 'size' )
javax.swing.JButton
| javax.swing.AbstractButton
| | javax.swing.JComponent
| | | java.awt.Container
| | | | java.awt.Component
* * * * ancestorResized, baselineResizeBehavior * * * * componentResized,
maximumSize, maximumSizeSet * * * * minimumSize, minimumSizeSet,
preferredSize * * * * preferredSizeSet, size
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver
| | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | java.io.Serializable
| | java.awt.ItemSelectable
| | javax.swing.SwingConstants
| javax.accessibility.Accessible
wsadmin>
```

What are all of those “size” attributes in the java.awt.Component class, anyway? Well, the size attribute identifies the current width and height of the component, which shouldn’t be too much of a surprise. The others—minimumSize, maximumSize, and preferredSize—are suggestions, or hints,[9] to the Layout

Manager about how the particular component should be represented.

The effect of the minimumSize attribute can be the borne out with a little testing. Figure 5-10 shows the initial application image, including the separators between the split panes. Each button in the application uses the same event handler routine to display the button text, the current button size, and the other size attributes mentioned previously. I encourage you to use the application to improve your understanding about the various component size attributes.

[6] See http://en.wikipedia.org/wiki/Guns_versus_butter_model.
[7]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JComponent.html.
[8]See http://docs.oracle.com/javase/8/docs/api/java/awt/Component.html.
[9]"…the code is more what you'd call 'guidelines' than actual rules." From http://www.imdb.com/title/tt0325980/quotes.

Figure 5-10. *SplitPane4: initial rendering*

To use the application, you need only click on the buttons to display the button size attributes. Then you can resize the window and use the buttons again to see which attribute values have changed. When the window size increases, you can also move the split pane divider lines by dragging them. Again, you can use the buttons to see the effect on the button size attributes.

Nested Split Panes
The application that illustrates the divider movement seen in Figure 5-10 uses the simple concept of nested split panes. Listing 5-12 shows the part of the SplitPane4.py script that produced this output.
Listing 5-12. Nested Split Panes

The button() method, on lines 17, 20, and 21, is a local reference to a method used to create a JButton with the actionPerformed() method assigned. The actionPerformed() method is the event handler used to display the button component sizes.

One interesting thing from this listing is how the top component (i.e., the second argument to the JSplitPane constructor) in line 17 is a simple call to the JButton constructor. Notice how the next argument, (i.e., the third for the JSplitPane constructor), which is used to specify the bottom component for the vertically split pane, is itself a JSplitPane constructor. This is the nesting to which this

section refers.

Imagine how much fun you could have using this technique to create deeply nested split pane components![10] If you search the web, you can find an interesting article, written by Hans Muller entitled “MultiSplitPane: Splitting Without Nesting.”[11] If you are really adventurous, you might investigate adding his class files to your environment and using them or converting them to Jython. If you do, please let me know how it goes.

Divider and Conquer

What, exactly, can you do with, and about, the separator (aka divider) bar? As you saw earlier, the Layout Manager decides how to allocate the available space to display the components. Do you have any control over the divider bar, and how it can be used? Sure, there are a number of JSplitPane methods related to the divider.

For example, Figure 5-11 shows how you can easily adjust the size of the divider bar using the setDividerSize() method. The image corresponds to the selected size button (i.e., the top image has a divider size of zero, the middle image has a divider size of 10, and the bottom image has a divider size of 20).

[10]This was discussed in Chapter 3.

[11]See http://today.java.net/pub/a/today/2006/03/23/multi-split-pane.html. **Figure 5-11.** *SplitPane5: various divider sizes*

One SplitPane attribute with which you may be unfamiliar is the OneTouchExpandable property. By default, this Boolean attribute is set to false. When it is true, the divider has two little triangles added to it, as shown in Figure 5-12.

Figure 5-12. *SplitPane6 with the **OneTouchExpandable** divider*

By enabling OneTouchExpandable, you allow the user to minimize one component with a single mouse click, which is really quite cool. Looking at the script used to produce this output, which can be found in the SplitPane6.py script file, you can see that you only needed to specify the oneTouchExpandable

keyword argument on the JSplitPane constructor call. Listing 5-13 shows how easily this can be accomplished using Jython.

Listing 5-13. oneTouchExpandable Keyword Argument

```
15| frame.add(
16| JSplitPane(
17| JSplitPane.HORIZONTAL_SPLIT,
18| JButton( 'Left' ),
19| JButton( 'Right' ),
20| oneTouchExpandable = 1
21| )
22| )
```

Rules for Using Split Panes

Unlike a during knife fight, there are rules for using a split pane. Most of them deal with how to determine the component sizes, and therefore the divider bar. This can also lead to dealing with the sizes of any nested components, which can be a challenge. In fact, there is a wonderful statement on the Java tutorials page entitled “How to Use Split Panes”[12] that says:

■**Note** Choosing which sizes you should set is an art that requires understanding how a split pane’s preferred size and divider location are determined.

Immediately after this statement, the page contains a bulleted list of rules for making split panes work well for you and your applications. If you intend to use split panes, I suggest that you bookmark that URL and study it often. I certainly have.

Can I Run a Tab? Using a TabbedPane

This section discusses something else that really isn’t a Layout Manager. It seems to be related to the CardLayout Layout Manager you read about earlier. In fact, it’s a different kind of panel. The reason I am discussing it here is because of the similarities it has to the CardLayout Manager mentioned in the previous section. Figure 5-13 shows how JTabbedPane can be used to display one of a group of panes based on the user’s selection. Unlike the CardLayout Layout Manager, an instance of JTabbedPane has a selectable tab label that can be used to determine the content to be displayed.

[12]Seehttp://docs.oracle.com/javase/tutorial/uiswing/components/splitpane.html.

Figure 5-13. *TabbedPaneDemo examples*

To demonstrate the similarities and differences between a panel using the CardLayout Manager and a TabbedPane class, the code shown in Listing 5-14 uses the same child panel contents as those seen in Listing 5-8.

Listing 5-14. TabbedPaneDemo Class

```
class TabbedPaneDemo( java.lang.Runnable ) :
 def run( self ) :
```

```
12| frame = JFrame(
13| 'TabbedPaneDemo',
14| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
15| )
16| self.addTabs( frame.getContentPane() )
17| frame.setSize( 300, 125 )
18| frame.setVisible( 1 )
19| def addTabs( self, container ) :
20| tab1 = JPanel()
21| tab1.add(
22| JLabel(
23| 'The quick brown fox jumped over the lazy dog.' 24| )
25| )
26| tab2 = JPanel()
27| for name in 'A,B,C'.split( ',' ) :
28| tab2.add( JButton( name ) )
29| tab3 = JPanel()
30| tab3.add(
31| JLabel(
32| 'Now is the time for all good men to come to...' 33| )
34| )
35| tabs = JTabbedPane()
36| tabs.addTab( 'Uno' , tab1 )
37| tabs.addTab( 'Dos' , tab2 )
38| tabs.addTab( 'Tres', tab3 )
39| container.add( tabs )
```

What’s the main difference? Well, in lines 36-38, you can see how the child panes are added to the JTabbedPane instance using the addTab() method instead of the JPanel add() method shown in lines 49-51 of Listing 5-8.
A JTabbedPane instance is also easier because the selection mechanism is built into the class. Take another look at Listings 5-7 and 5-9, where you had to specifically identify how the CardLayout panel instances would be selected by the users.

Are You Boxed In? Using the BoxLayout Manager

Earlier in this chapter, you saw how easy it was to use the FlowLayout Layout Manager to position the components in a row. You also saw that when the

application window was resized, the Layout Manager would reposition the components on different rows when the width of the application window would otherwise be too narrow. There are times when you don't want this to occur. This leads to the next Layout Manager, i.e., the one defined in the BoxLayout class.[13]

Figure 5-14 shows the sample output of the BoxLayoutDemo sample application. It uses a JTabbedPane with three tabs. In each tab, you can see three buttons that are layered out along the Y_AXIS (i.e., vertically). The only difference between the tabs is how the buttons are aligned within the pane.

[13]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/BoxLayout.html.

BoxLayoutDemo
Left
Center
Right
1
2
3 being the third number

BoxLayoutDemo
Left
Center
Right
1
2
3 being the third number

BoxLayoutDemo
Left
Center
Right
1
2
3 being the third number

Figure 5-14. *BoxLayout tabs*

Listing 5-15 shows the BoxLayoutDemo class that produces the output shown in Figure 5-14. As you can see, when you instantiate the Layout Manager, as shown in line 29, you tell it how the components should be displayed. Later, in line 31, you see that when you instantiate the buttons, you can use a Component alignment constant to tell the component where the other part should be displayed (i.e., along the x-axis of the panel).

Listing 5-15. BoxLayoutDemo Class

```
10|class BoxLayoutDemo( java.lang.Runnable ) :
11| def run( self ) :
12| frame = JFrame(
13| 'BoxLayoutDemo',
14| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
15| )
16| self.addTabs( frame.getContentPane() )
17| frame.setSize( 300, 175 )
18| frame.setVisible( 1 )
19| def addTabs( self, container ) :
20| align = [
21| [ 'Left' , Component.LEFT_ALIGNMENT ],
22| [ 'Center', Component.CENTER_ALIGNMENT ],
23| [ 'Right' , Component.RIGHT_ALIGNMENT ]
24| ]
25| names = '1,2,3 being the third number'.split( ',' )
26| tabs = JTabbedPane()
27| for aName, aConst in align :
28| tab = JPanel()
29| tab.setLayout( BoxLayout( tab, BoxLayout.Y_AXIS ) )
30| for name in names :
31| tab.add( JButton( name, alignmentX = aConst ) )
32| tabs.addTab( aName, tab )
33| container.add( tabs )
```

The constants provided by the BoxLayout class include those listed in Table 5-1.

Table 5-1. *BoxLayout Constants* **Constant**

BoxLayout.X_AXIS BoxLayout.Y_AXIS BoxLayout.LINE_AXIS

BoxLayout.PAGE_AXIS

Description
Components are laid out horizontally, left to right.
Components are laid out vertically, top to bottom.

Components are laid out based on the container's ComponentOrientation property, either horizontally or vertically.
Components are laid out the way text is displayed on a page, based on the container's ComponentOrientation property, either horizontally or vertically.
The Box Class

Even though this chapter has been focusing mainly on Layout Managers, you have also learned about some container classes to help define how the application components are displayed. The Box class is another of these lightweight containers that helps arrange application components.

If you need to create a single row or column of components, you might want to take a look at the Box class,[14] which works kind of like a JPanel. One significant difference between these two classes is that the Box class can *only* use a BoxLayout Layout Manager,[15] whereas the JPanel allows the developer to choose any kind of Layout Manager.[16]

Building a Box

The Box class has a single constructor, as shown in Figure 5-15. One of the interesting things about this constructor is that it isn't used as often as the createHorizontalBox() and createVerticalBox() methods. I think that calling the Box.createHorizontalBox() method is more understandable than Box(BoxLayout.X_AXIS), don't you? Can you think of a reason that you might prefer the constructor to the createHorizontalBox() method?

public Box(int axis)
Creates a Box that displays its components along the specified axis.
Parameters:
axis - can be BoxLayout.X_AXIS, BoxLayout.Y_AXIS, BoxLayout.LINE_AXIS or BoxLayout.PAGE_AXIS.
Throws:
AWTError -if the axis is invalid

See Also:
createHorizontalBox(), createVerticalBox()
Figure 5-15. *Box constructor*

If nothing comes to mind, you might want to take another look at Table 5-1. The important thing to note is that the createHorizontalBox() class is equivalent to using the Box(BoxLayout.X_AXIS) constructor call, which will position the components left to right, regardless of the user locale. This might not be what you intend. You have to be the judge.

[14] Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/Box.html.
[15]See http://docs.oracle.com/javase/8/docs/api/javax/swing/Box.html#setLayout%28jav
[16]By default, a JPanel instance will use a FlowLayout Layout Manager.

Invisible Box Components

One of the challenges with using a Box instance to hold your components is the fact that they are, by default, adjacent to one another. Figure 5-16 shows what happens when three fixed-size components (i.e., JLabel instances) are added to a horizontal box container. The left image shows how the application looks when the application is first made visible. The right image shows what happens to the components when the application is widened.

Figure 5-16. *Horizontal (fixed-size) components*
If you were expecting the components to be centered or right-aligned, you aren't going to be pleased with the results. To help adjust this kind of alignment, the Box class includes the following "filler" components:

- Glue
- Struts
- Rigid areas

Your choice of filler component will depend upon the initial appearance of your box, as well as on how you want it to look when it is resized.
Glue

If you research Java Swing glue components, most of the references mention the fact that glue is not a very good description for this type of component. This is true because most people think of glue as a kind of adhesive. Instances of this invisible Swing component will start out with a width or height of zero. This attribute then increases in size as the container size increases.

Three Box methods exist that can be used to create glue components. These methods are listed and described in Table 5-2.

Table 5-2. *Box Class Glue Methods* **Method Name** createGlue()

createHorizontalGlue() createVerticalGlue()
Description/Details
Used to create a component that can be used between horizontal or vertical components.
Used to create a component that is oriented horizontally.
Used to create a component that is oriented vertically.

Glue components have a minimum size (horizontal width or vertical height) of zero. This means that it shouldn't be obvious when a glue component is between other components on the box container. When the container becomes larger, the Layout Manager will increase the size of all of the resizable components (i.e., the ones smaller than their maximum size). When multiple fixed-size components are positioned between glue components, the available space will be allocated equally to the glue components.

It may help to see an example to better understand what this means. Table 5-3 shows what happens when glue components are positioned around some fixed-size components (i.e., JLabel instances), and the application frame containing the horizontal box is widened.

Table 5-3. *Using Glue Components in a Horizontal Box*

horizontalBox
<---- Left ----> <---- Middle ----><---- Right ---->
horizontalBox
<---- Left ----> <---- Middle ----> <---- Right ---->
horizontalBox
<---- Left ----> <---- Middle ----> <---- Right ---->
No glue components exist, so all the “extra” space is allocated to the right of the last label.
One glue component is added before the first label, so it gets all the “extra” space.

Glue components are added before and after the first label. The “extra” space is equally distributed between the glue components.

Glue components are added in three places. The “extra” space is equally distributed among these glue components.

Glue components are added before, after, and in between the labels. All the “extra” space is allocated equally.
When a vertical box is used to hold components, glue components act in a similar fashion.
Strut Components
There are times when it is appropriate to have some minimum distance between components. For those times, consider using horizontal or vertical struts. Table 5-4 shows two methods that you can use to create these strut components.

Table 5-4. *Box Class Strut Methods*
Method Name Description/Details
createHorizontalStrut(int width) Create a filler component of the specified width (in pixels). createVerticalStrut(int height) Create a filler component of the specified height (in pixels).

Is there any problem related to using strut components? Well, I have found some references that indicate that horizontal strut components have an unlimited height, and vertical strut components have an unlimited width, which can be a

problem. Apparently, if you use strut components with nested vertical and horizontal box containers, alignment issues can arise. Take a look at this example to see what they are talking about. Figure 5-17 shows an application that has five buttons (all of which have the same text) aligned in a plus pattern. The issue isn't obvious until you drag the bottom-right corner of the application down to increase the size and width. As you can see from the image on the right, the results are unexpected. The glue components are placed before the leftmost button, after the rightmost one, above the top one, and beneath the bottom one. When the frame becomes larger, the strut component before the rightmost button has extra space.

Figure 5-17. *Alignment issues with strut components*

Rigid Area Components

The final invisible component—called a rigid area—is created using specific, fixed dimensions. This means that the component size should not change even if its container is resized. This would seem to make it the preferred filler component if your application is best rendered with some space between visible components. Table 5-5 shows the Box method that you can use to create this type of invisible component.

Table 5-5. *Rigid Area Creation Method*

Method Name	Description/Details
createRigidArea(Dimension d)	Creates an invisible component that's always the specified size.

Boxes and Resizable Components

Up to this point, you have been using fixed-size components when dealing with

box containers (e.g., labels and buttons). It is important to note, however, that some components have a maximum size that isn’t the same as the minimum and preferred size. What happens when that occurs?

Based on what I have already discussed, you probably realize that the Layout Manager will also allocate available space to these components as space becomes available. The distribution of the available size depends on a number of factors, including:

• The maximum size attribute of the resizable components
• The current size of the container
• The size of the screen

I don't know about you, but I was a little surprised about this last one. In fact, I had to try it out, so I played around with the fBox.py script, which displays information about the text field width when the button is pressed. When the maximum width of the text field is less than or near the screen width, the amount of space allocated to the text field as the application width increases is relatively small. As the maximum width of the text field is increased, you’ll see that the amount of space allocated to the text field as the application is widened increases accordingly. If the maximum width of the text field is not limited (i.e., if the default maximum width is used), the text field will receive all of the available extra width.

Gridlock, Anyone? Using the GridLayout Manager

Another common layout configuration is when the components are laid out in a grid. The GridLayout Layout Manager provides this configuration. Figure 5-18 shows a simple example where the application has two panes, each of which contains a group of buttons displayed in a grid. Each grid has three columns and enough rows to display all of the buttons.

GridLayoutDemo
Horizontal: 0
Horizontal: 2
Horizontal: 4
Horizontal: 8
Horizontal: 16
Vertical: 0
Vertical: 2
Vertical: 4
Vertical: 8
Vertical: 16

GridLayoutDemo
Horizontal: 0
Horizontal: 2
Horizontal: 4
Horizontal: 8
Horizontal: 16
Vertical: 0
Vertical: 2
Vertical: 4
Vertical: 8
Vertical: 16

GridLayoutDemo
Horizontal: 0
Horizontal: 2
Horizontal: 4
Horizontal: 8
Horizontal: 16
Vertical: 0
Vertical: 2
Vertical: 4
Vertical: 8
Vertical: 16

Figure 5-18. *GridLayoutDemo examples*

The interesting thing about this application is that the buttons allow you to dynamically change the horizontal or vertical gap between the buttons on each pane that is using the GridLayout Layout Manager.
Listing 5-16 shows the run() method in the GridLayoutDemo application. In this method, you can see how the main panel uses a BoxLayout Layout Manager to position two inner panes along the Y_AXIS (i.e., vertically). Then, the addButtons() method is called to add another inner pane, which uses a GridLayout Layout Manager to position a group of buttons.

Listing 5-16. GridLayoutDemo run() Method

```
def run( self ) :
frame = JFrame(

'GridLayoutDemo',
defaultCloseOperation = JFrame.EXIT_ON_CLOSE
)
main = JPanel()
main.setLayout( BoxLayout( main, BoxLayout.Y_AXIS ) )
self.panes = []
self.addButtons( main, 'Horizontal:' )
self.addButtons( main, 'Vertical:' )
frame.add( main )
frame.setSize( 500, 250 )
frame.setVisible( 1 )
```

Listing 5-17 shows how the addButtons() method creates a new panel, populates it with buttons, and then adds it to the specified container. A reference to the new panel is saved in the self.panes list, created in the run() method, for use by the button event handler.

Listing 5-17. GridLayoutDemo addButtons() Method

```
def addButtons( self, container, prefix ) :
pane = JPanel( GridLayout( 0, 3 ) )
self.panes.append( pane )
for size in '0,2,4,8,16'.split( ',' ) :

pane.add(
```

```
JButton(
'%s %s' % ( prefix, size ),
actionPerformed = self.buttonPress

)
)
container.add( pane )
```

Listing 5-18 shows how the button label is retrieved using the event.getActionCommand() method. The program then uses the space that exists between the direction (i.e., Horizontal or Vertical) and the size (i.e., the number of pixels) for the inner component gap.

Listing 5-18. GridLayoutDemo buttonPress() Method

```
def buttonPress( self, event ) :
dir, size = event.getActionCommand().split( ' ' ) if dir[ 0 ] == 'H' :

for pane in self.panes :
layout = pane.getLayout()
layout.setHgap( int( size ) )
layout.layoutContainer( pane )

else :
for pane in self.panes :
layout = pane.getLayout() layout.setVgap( int( size ) ) layout.layoutContainer(
pane )
```

This is where you need to process each of the panes using a GridLayout Layout Manager. Given a pane, you can obtain a reference to the Layout Manager used by that pane using the pane.getLayout() method call. You can use this reference to adjust the horizontal or vertical gap used by this Layout Manager (i.e., by calling the setHgap() or setVgap() method).

Once the gap has been changed, a call is made to the layoutContainer() method to force the Layout Manager to adjust how the components are displayed.

Shaking Things Up: The GridBagLayout Manager

One of the problems with the GridLayout Manager is that sometimes you’ll want

there to be some differences between the size and placement of the components. For example, you might want things to be arrayed in rows, but not use the regular column arrangement provided by GridLayout. You can use the GridBagLayout Layout Manager for these purposes. Unfortunately, it comes with a price. Specifically, in order for you to have more control over the size and placement of components in a grid, you must provide a greater level of detail to each component. Therefore, the GridBagLayout Manager is a bit more complicated than the others.

Figure 5-19 shows sample output of the GridBagLayout applications found in the code\Chap_05 directory. These are just examples of the kind of control that the GridBagLayout Layout Manager provides for component placement.

Figure 5-19. *GridBagLayout examples*

Listing 5-19 shows the addComponents() method from the GridBagLayout4 script, which was used to generate the output shown in Figure 5-19. There are some important points about this method to note. For example, a new GridBagConstraints instance is created for each component. Why is this considered a “best” practice? This is recommended because the reuse of existing GridBagConstraints objects can easily lead to subtle, and therefore difficult-to-

diagnose, problems. It is all too easy to forget that one of the many constraint fields has a non-default value.

Listing 5-19: GridBagLayout4 addComponents() Method

```
def addComponents( self, container ) :
container.setLayout( GridBagLayout() )
c = GridBagConstraints()
c.gridx = 0 # first column
c.gridy = 0 # first row
container.add( JButton( '1' ), c )
c = GridBagConstraints()
c.gridx = 1 # second column
c.gridy = 1 # second row
container.add( JButton( '2' ), c )
c = GridBagConstraints()
c.fill = GridBagConstraints.HORIZONTAL
c.gridx = 2 # third column
c.gridy = 2 # third row
c.weightx = 0.0
c.gridwidth = 3
container.add( JButton( '3 being the third number' ), c ) c = GridBagConstraints()
c.gridx = 1 # second column
c.gridy = 3 # forth row
c.ipady = 32 # make this one taller
container.add( JButton( 'Four shalt thou not count' ), c ) c =
GridBagConstraints()
c.gridx = 1 # second column
c.gridy = 4 # fifth row
c.gridwidth = 3 # make this one 3 columns wide container.add( JButton( 'Five is
right out' ), c )
```

This provides me with another opportunity to highlight some of the features and strengths of Jython. If you take a look at the Javadoc for the GridBagConstraints class,[17] you will see that two constructors exist. The first constructor has no parameters and creates a GridBagConstraints instance using all of the default values. The other has 11 parameters and requires that all of the values be provided.

Jython, on the other hand, lets you use keyword arguments to selectively provide only those values of interest. Listing 5-20 shows an interactive session that uses a displayConstraints() function[18] to show any non-default constraint values that exist in the specified object. In lines 8, 14, and 19, you can see how keyword arguments can be provided when the object is instantiated.

[17]Seehttp://docs.oracle.com/javase/8/docs/api/java/awt/GridBagConstraints.html.
[18]The complete source for which can be found in code\Chap_05\displayConstraints.py script file. **Listing 5-20.** Specifying GridBagConstraints parameters using keywords

```
wsadmin>from java.awt import GridBagConstraints
wsadmin>from java.awt import Insets
wsadmin>
wsadmin>c = GridBagConstraints()
wsadmin>displayConstraints( c )
All constraint values match defaults
wsadmin>
wsadmin>c = GridBagConstraints( gridx = 1, gridy = 2 )
wsadmin>displayConstraints( c )
Non-default constraint values:

gridx: 1
gridy: 2
wsadmin>
wsadmin>c = GridBagConstraints( insets = Insets( 1, 2, 3, 4 ) )
wsadmin>displayConstraints( c )
Non-default constraint values:
insets: java.awt.Insets[top=1,left=2,bottom=3,right=4] wsadmin>
wsadmin>c = GridBagConstraints( fill = GridBagConstraints.CENTER )
wsadmin>displayConstraints( c )
Non-default constraint values:
fill: CENTER
wsadmin>
```

Looking at Other Layout Managers

In addition to the Layout Managers you already read about, there is the

SpringLayout class[19]. I'm not going to cover it here because of its complexity, as well as because it's most often used by GUI builders. At this time, I am unaware of any GUI builder that generates Jython code, so it would seem that your best bet is to find one that generates Java code and translate it to the corresponding Jython code. Unfortunately, I don't know how many people would invest the time to do this, especially if the application's "look and feel" changes frequently.

Another Layout Manager, called GroupLayout,[20] exists, and is also used by GUI builders. It too is beyond the scope of this book due to its complexity. If you are interested in learning more about it, read the section in the Java Swing tutorials called "How to Use GroupLayout."[21]

The other alternative that isn't covered here is the "roll your own" option. There is good information about this approach at your fingertips; see the section in the Java Swing tutorials entitled "Creating a Custom Layout Manager."[22] This, too, is beyond the scope of this book, and I don't have enough experience with it to feel comfortable trying to tackle this topic here.

[19] Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/SpringLayout.html.
[20]See http://docs.oracle.com/javase/8/docs/api/javax/swing/GroupLayout.html.
[21]See http://docs.oracle.com/javase/tutorial/uiswing/layout/group.html.
[22]See http://docs.oracle.com/javase/tutorial/uiswing/layout/custom.html.

Summary

This chapter is all about providing you with information to help you understand how you can use Layout Managers or containers to position your application components. As you can see, there are a number of options open to you and your applications. It's all about how you want the application components to be arranged. One of the aspects discussed in each section is how the components change when the container is resized. This is an important topic to consider when you are designing and developing your application, so keep that in mind.

Chapter 6

Using Text Input Fields

Up to this point, you've been limited in the kinds of applications that you can

build because of the lack of ways of getting information (i.e., input) into the application. You will now start to remedy that deficiency by learning to create an input field that lets the users enter text.

This chapter discusses a couple of input-related fields such as JTextField and JTextArea. Additionally, you will take a look at how the data is displayed e.g., using alignment and fonts. The chapter also covers creating a simple Swing application that allows you to display and modify a value in the WebSphere Application Server (WSAS) environment. One reason to do this is so that you can see how to perform “long-running” operations on a separate thread using instances of the SwingWorker class.

What Does It Take to Get Data Into an Application?

To start, you’ll create a fairly simple value that you want to be able to view and possibly modify. How about the WebSphere Administrative Console inactivity timeout? Well, unfortunately, the WebSphere documentation only contains one sample Jacl script that shows how this can be done.[1] You’ll start by creating two Jython functions, one to get the current inactivity timeout value and the other to set it.

Listing 6-1 shows an interactive wsadmin session where these functions are used to get and set the timeout value. The complete functions can be found in code\Chap_06 directory.[2] They’ve been tested, as shown in Listing 6-1, so you should be all set to use them, right?

Listing 6-1. Getting and setting the admin console inactivity timeout

```
wsadmin>print getTimeout()
30
wsadmin>print setTimeout( '123' ) Successfully modified.
wsadmin>print getTimeout()
123
wsadmin>print setTimeout( '30' ) Successfully modified.
wsadmin>
```

[1] See http://www14.software.ibm.com/webapp/wsbroker/redirect?version=compass&product=was-nddist&topic=cons_sessionto.

[2] In files named getTimeout.py and setTimeout.py, respectively.

JTextField: Getting Data Into the Application

To make things easy, you will start with a simple text field that solicits information from the users. What is required and how does it work? You start by creating a JTextField instance, which provides a simple, lightweight input field that allows users to enter a small amount of text. When the user input is complete, which is generally indicated by the user pressing the Enter key, any associated ActionListener will receive an action event.[3]

If you look at the Java documentation for the JTextField class,[4] you'll see that a number of constructors exist. The simplest of these requires no parameters. One of those enables you to provide an initial string value to be displayed, one allows the field width in number of columns, and another allows the initial string value to be specified as well as the number of columns to be displayed. This application will use something simple—the one that allows you to specify the number of columns to be displayed. Since you are unlikely to need more than three digits, you can use a value of three for the number of columns.

To provide the users with visual indications of what the input field contains, you need to surround it with some labels. And to be a little more complete, you can have a message label field on a separate line to display a message indicating the success or failure of the requested action.

■**Note** do you remember how you were able to use the actionPerformed keyword assignment as part of the constructor call when you created a JButton? if not, have a look at Chapter 4.[5]
You can use the same technique to specify the ActionListener routine to be called when an actionEvent occurs. In the case of JTextField, this is when the user presses the Enter key.

Your First, Almost Real, Application

What does it take to use one or more existing routines and turn them into a graphical wsadmin application? Well, after a bit of work, you're likely to take what you have learned, including the information about the JTextField, and build an application, the output of which might look something like Figure 6-1.

Figure 6-1. *consoleTimeout1 sample output*

[3] I haven't covered action events in great detail, at least not yet. They are covered in Chapter 14. However, you have used the actionPerformed() method when you learned about buttons earlier.

[4]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JTextField.html.

[5]For example, take a look at Listing 4-8.

What code do you need to do this? Listing 6-2 shows the consoleTimeout1 class from the sample application script file. Note how it calls the getTimeout() and setTimeout() routines, just like the interactive session shown in Listing 6-1. This makes it look pretty simple, doesn't it?

Listing 6-2. consoleTimeout1 Class

```
class consoleTimeout1( java.lang.Runnable ) : def run( self ) :
frame = JFrame(
'Console timeout',
defaultCloseOperation = JFrame.EXIT_ON_CLOSE )
cp = frame.getContentPane()
cp.setLayout( BoxLayout( cp, BoxLayout.Y_AXIS ) )

input = JPanel( layout = FlowLayout() )
input.add( JLabel( 'Timeout:' ) )
self.text = JTextField( 3, actionPerformed = self.update ) input.add( self.text )
self.text.setText( getTimeout() )
input.add( JLabel( 'minutes' ) )
cp.add( input )

self.msg = cp.add( JLabel() )
frame.setSize( 290, 100 ) frame.setVisible( 1 )

def update( self, event ) :
value = self.text.getText().strip()
if re.search( '^\d+$', value ) :
```

```
self.msg.setText( setTimeout( value ) ) else :
msg = 'Invalid value "%s" ignored.' % value
self.msg.setText( msg )
self.text.setText( getTimeout() )
```

Unfortunately, it isn't as simple as it looks. If you test the application and watch closely, you are likely to notice a delay between the time that you type a value and press Enter, and the message being displayed. What's happening?
Welcome to the world of event-driven applications. What you are seeing is the fact that a non-trivial delay is occurring between the time that the setTimeout() routine is being called and when it returns. In the meantime, the entire application is hung. This is a really bad thing, and exhibits a terrible practice.

Help Me SwingWorker, You're My Only Hope!

A first attempt at fixing this might involve changing the message field to show some kind of message. You might even want to disable the input field before calling the setTimeout() routine. So, how might you do this? Well, you could replace the call to the setTimeout() routine in Listing 6-1 with something like what's shown in Listing 6-3. **Listing 6-3.** Attempt to Fix the update() Method

```
self.msg.setText( 'working...' ) self.text.setEnabled( 0 )
self.msg.setText( setTimeout( value ) ) self.text.setEnabled( 1 )
```

What do the calls to the setEnabled() method associated with the JTextField do? The first one attempts to disable it by specifying 0, which is interpreted as false, and the second call attempts to enable it by specifying 1, which is interpreted as true.[6]

If it worked, this would disable the input field so that the AdminConfig scripting object could locate and modify the inactivity timeout value associated with the appropriate configuration object.
Unfortunately, this doesn't work because the part of the Swing framework that updates the screen never has a chance to gain control to update the display. How do you fix this? How do you force the application to perform some separate action while the Swing framework displays the GUI and allows the user to generate events like button clicks?
Well, to do this, I need to take a trip down the rabbit hole and talk about something called *concurrency*.[7] It's similar to when you have multiple separate

programs running on your computer at the same time. In this case, you need the application to have multiple things going on. The challenge is that all of this needs to be happening within one program, the wsadmin Jython script.

When multiple programs execute at the same time, the operating system and the program developer are responsible for keeping things straight, so that two or more programs don't try to manipulate the same piece of information (e.g., an object or variable) at the same, or nearly the same time.

How do you have separate things going on at the same time in your Jython scripts? Well, you are going to have to make use of a concurrent programming concept called *threads*. You need to identify operations that may require a "long" time to complete and have these executed on a separate thread. A *thread* is a short form of, or nickname for, a thread of control. Each thread operates separately and distinctly.

Since I don't have the time or space to completely cover this topic, I am going to try to provide just enough so that you can resolve the common issues that you'll likely encounter when using the applications discussed.

The really good news is that Jython scripts are executing on a Java Virtual Machine, and Java was created with concurrency in mind. This means that your scripts can make use of classes and techniques that have been part of Java for years.

The primary class that you need to use is called SwingWorker. The Javadoc for the SwingWorker class[8] includes a simple case example that translates to Listing 6-4.

[6] Just like you do with the call to frame.setVisible(1) in the application run() method.

[7]Concurrent programming is a large topic all by itself. I don't have the time or space to cover it here completely. You will, however, learn enough to be able to develop graphical Jython applications. There are lots of good articles available on the topic. I encourage you to search for "threads and concurrent programming."

[8]See http://docs.oracle.com/javase/8/docs/api/javax/swing/SwingWorker.html.

Listing 6-4. Simple SwingWorker Subclass

```
class InTheSwing( SwingWorker ) :
def __init__( self, labelField = None ) : self.label = labelField
def doInBackground( self ) :
try :
```

```
self.result = longRunningRoutine() except :
self.result = 'Exception encountered.' def done( self ) :

self.label.setText( self.result ) ...
label = frame.add( JLabel() )
InTheSwing( label ).excecute()
```

How well does this work? The images shown in Figure 6-2 can answer this question. You can see that when there's an invalid value entered (in this case "x"), the bad value is replaced with the original value and an appropriate error message is displayed. The next image shows what happens when a valid value is entered. The input field has been disabled and a "working..." message is displayed, at least until the update is complete. The last image shows that the input field has been enabled; the "Update successful" message is displayed.

Figure 6-2. *consoleTimeout2 Sample Output*

What code do you need to do this? Listing 6-5 shows the WSAStask class, which is a SwingWorker descendent class. How does it work?

Listing 6-5. WSAStask Class

```
58|class WSAStask( SwingWorker ) :
59| def __init__( self, textField, labelField ) :
60| self.text = textField # Save the References
61| self.label = labelField
62| SwingWorker __init__( self )
63| def doInBackground( self ) :
64| self.text.setEnabled( 0 ) # Disable input field
65| self.label.setText( 'working...' ) # Inform user of status
66| value = self.text.getText().strip()
```

67| if not re.search(re.compile('^\d+$'), value) : 68| msg = "Invalid numeric value "%s",' % value 69| self.label.setText(msg)
70| self.text.setText(getTimeout())
71| else :
72| self.label.setText(setTimeout(value)) 73| def done(self) :
74| self.text.setEnabled(1) # Enable input field

The explanation of how this class works can be found in Table 6-1. Fortunately, the code that needs to be executed on a separate thread of control is quite easy to understand.

Table 6-1. *WSAStask Class, Explained*
Lines Description/Explanation

59–62 Class constructor used to save references to the necessary component fields and call the SwingWorker (i.e., base class) constructor (i.e., the SwingWorker.__init__() method).

63–72 Method called by the SwingWorker execute() method that does the actual, possibly long, running work. It is interesting to note how calls to global functions (i.e., getTimeout() and setTimeout()) are made here. Note also how the field references saved by the constructor method (i.e., the __init__() method) are used here to access and manipulate the actual components.

73–74 The done() method is also called by the SwingWorker execute() method when the doInBackground() method completes.

What does that do for your consoleTimeout class? In Listing 6-2, the update() method required half a dozen steps. Listing 6-6 shows how you only need to instantiate a WSAStask object and call its execute method to do the work. Remember that all of the stuff that was previously done in the update() method has been moved to the WSAStask doInBackground() method. So the code to perform the work is now performed on the separate thread of control.

Listing 6-6. consoleTimeout2's update() Method def update(self, event) : WSAStask(self.text, self.msg).execute()
■**Warning** it's important to note that SwingWorker objects cannot be reused. additional work, even if it is identical to what was done previously, must be done using a completely new instance of the SwingWorker descendant class.

Back to the JTextField

Up to this point, I haven't provided many details about the JTextField component. The reason for this was to allow you to quickly use a simple kind of input field for the sample application. Now, however, it's appropriate to revisit the JTextField class, so you can garner a more complete understanding of its capabilities, limitations, and uses.

If you look into the Java Swing Tutorial, you'll can find a section entitled, "How to Use Text Fields"[9] Let's start by describing and using the JTextField, which is the simplest of the text input fields.
Looking at the JTextField Javadoc shows that it includes a number of methods. Unfortunately, I don't have the time or space to completely describe each and every method. However, there are some that warrant investigation. One getter/setter pair that is likely to catch your eye is the one dealing with horizontal alignment. Figure 6-3 shows the output of the TextAlignment.py[10] script, which illustrates the various JTextField alignment values and how they affect text display in a JTextField.

Figure 6-3. *TextAlignment output*

Listing 6-7 shows the TextAlignment class from the script used to display this output. It is interesting to note how easily you can display this information using a GridLayout. You don't even have to tell the Layout Manager how many rows will be displayed. By initializing the number of rows as 0, as shown on line 12, you let the Layout Manager keep track of how many rows are provided.

Listing 6-7. TextAlignment Class

```
8|class TextAlignment( java.lang.Runnable ) :
9| def run( self ) :
10| frame = JFrame(
```

```
11| 'TextAlignment',
12| layout = GridLayout( 0, 2 ),
13| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
14| )
15| data = [
16| [ 'Left' , JTextField.LEFT ],
17| [ 'Center' , JTextField.CENTER ],
18| [ 'Right' , JTextField.RIGHT ],
19| [ 'Leading' , JTextField.LEADING ],
20| [ 'Trailing', JTextField.TRAILING ]
21| ]
22| for label, align in data :
23| frame.add( JLabel( label ) ) 24| text = frame.add(
25| JTextField(
26| 5,
27| text = str( align ), 28| horizontalAlignment = align 29| )
30| )
31| frame.pack()
32| frame.setVisible( 1 )
```

[9]See http://docs.oracle.com/javase/tutorial/uiswing/components/textfield.html.
[10]The complete script is in the Code\Chap_06\TextAlignment.py file.

The other thing to note here is how you can use the Jython keyword parameter assignments (lines 12, 13, 27, and 28) to greatly simplify the code. Consider the difference between the Left and Leading alignments. Remember that you might be building applications that are used in a variety of locations across the world. Left and Right are absolute direction indicators, whereas the Leading and Trailing alignments are related to the locale and the direction that text should be displayed, based on the locale setting.

Size Matters: Looking at Text Font Attributes

While you’re looking at the JTextField documentation, it’s smart to consider also the setFont() method. Can you change the font used by the JTextField components? With a tiny bit of work, I was able to change the TextAlignment.py example to use a different font for each of the JTextField values. The output of this modification is shown in Figure 6-4.

Figure 6-4. *TextFonts output*

The changes you need to make to generate this output are shown in Listing 6-8. This isn't anything nearing a complete discussion about fonts; it's just a glimpse as to how easy they are to change.

Listing 6-8. TextFonts Changes

```
data = [
[ 'Left' , JTextField.LEFT , None ],
[ 'Center' , JTextField.CENTER , Font( 'Courier' , Font.BOLD, 12 ) ], [ 'Right' ,
JTextField.RIGHT , Font( 'Ariel' , Font.ITALIC, 14 ) ], [ 'Leading' ,
JTextField.LEADING , Font( 'Elephant', Font.BOLD | Font.ITALIC, 20 ) ], [
'Trailing', JTextField.TRAILING, Font( 'Papyrus' , Font.PLAIN, 36 ) ]

]
for label, align, font in data :
frame.add( JLabel( label ) )
text = frame.add(
JTextField(
5,
text = str( align ),
horizontalAlignment = align,
)
)
```

```
if font :
text.setFont( font )
```

This code adds an additional column for each row in the data array. The value of this column is None or a font instance. If a font instance is present, it is used to define the font to be used for the associated JTextField instance. The test and associated assignment are performed in the last two lines of Listing 6-6.

The Elephant (Font) in the Room

One of the questions that might come to you as you are looking at this code is, “Is there really a font named ‘Elephant’”? I was a little surprised by that one as well. This made me wonder what it would take to create a simple application that displays the list of available font names.

First, you need some kind of data area that can be used to hold a bunch of information. For this purpose, you’re going to use the next kind of input field, called a JTextArea. One of the things to note about this particular application is that it uses an output field, not an input field.

How do you do that? The Java documentation for this component[11] includes half a dozen constructors, half of which allow you to provide a string to be used to initialize the text. Listing 6-9 shows how you can obtain the list of available fonts from a local graphics environment instance. One thing that you have to note, however, is that you have to convert this list of strings into a single string, with newline characters delimiting each line of text. Fortunately, a simple Jython idiom exists to do exactly this, as you can see in line 19.

[11]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JTextArea.html.

Listing 6-9. AvailableFonts Class

```
8|class AvailableFonts( java.lang.Runnable ) :
9| def run( self ) :
10| frame = JFrame(

11| 'Available Fonts',
12| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
13| )
14| lge = GraphicsEnvironment.getLocalGraphicsEnvironment()
```

```
15| fontNames = lge.getAvailableFontFamilyNames()
16| frame.add(
17| JScrollPane(
18| JTextArea(
19| '\n'.join( fontNames ),
20| editable = 0,
21| rows = 8,
22| columns = 32
23| )
24| )
25| )
26| frame.pack()
27| frame.setVisible( 1 )
```

The other important thing to notice is that this text area is likely to contain more lines of text than will comfortably fit on the application screen. Therefore, you need to put the text area instance in a container that will automatically provide vertical and horizontal scroll bars, as needed, to comfortably display a reasonable application window. You do this using an instance of the JScrollPane class.[12]

Is it always this simple and easy? Well, it all depends. In this case, you can use almost all of the default settings. The other really good thing about Jython is that you can also use keyword arguments to help document what the parameters mean. So, all in all, you benefit from the well-designed Swing classes hierarchy.

Take a moment and comment out lines 21 and 22. Don’t forget to remove or comment out the trailing comma on line 20 as well.[13] Before you execute the script, what do you expect it to display? When I ran the modified script, it didn’t match my expectations. I don’t know about you, but I think that the use of the keyword arguments on the JTextArea constructor call certainly made the application better looking. This just to point out to you that not all of the Swing default values will match your expectations or needs. Try to keep this in mind.

Using JTextArea for Input

Let’s talk for a moment about the editable keyword assignment in line 20 of Listing 6-7. By now, you should be able to recognize it as equivalent to a call to the setEditable() method with a value of false. All this so the area of text can’t be

modified by the user. All you have to do to make this input component editable is remove this keyword assignment or change the value from 0 (false) to 1 (true).

What happens if you execute the script after making this change? Well, you can select, modify, remove, or add text. And all it really takes is a JTextArea instance. With the JTextField, as you saw earlier in the chapter, you can add an ActionListener in order for an event handler routine of your choice to be invoked when the user presses Enter.

[12] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JScrollPane.html.
[13]You could replace lines 16–25 with the following statement, which I find much easier to read: frame.add(JScrollPane(JTextArea('\n'.join(fontNames), editable = 0)))

Which listeners make sense for a JTextArea? I don't know. Let's use the classInfo() routine, seen earlier, to find out what kind of listeners can be added. Oops. That's one of the problems about asking questions. You may not want to see the answer you get. What are all of these things? Table 6-2 briefly explains the kinds of listeners that can be added.

Table 6-2. *JTextArea Listeners* **Listener Name**
AncestorListener
CaretListener
ComponentListener
ContainerListener
FocusListener
HierarchyBoundsListener HierarchyListener
InputMethodListener
KeyListener
MouseListener

MouseMotionListener MouseWheelListener PropertyChangeListener
VetoableChangeListener

Listener Description
Support notification when changes occur to a JComponent or one of its ancestors. Listens for changes in the caret position of a text component.
The listener interface for receiving component events.
The listener interface for receiving container events.

The listener interface for receiving keyboard focus events on a component. The listener interface for receiving ancestor moved and resized events. The listener interface for receiving hierarchy changed events.
The listener interface for receiving input method events.
The listener interface for receiving keyboard events (keystrokes).

The listener interface for receiving “interesting” mouse events (press, release, click, enter, and exit) on a component.

The listener interface for receiving mouse motion events on a component. The listener interface for receiving mouse wheel events on a component. A PropertyChange event gets fired whenever a bean changes a “bound” property.

A VetoableChange event gets fired whenever a bean changes a “constrained” property.

If you chose to have a listener for each kind of event, you would be able to micromanage the JTextArea instance. I’m not even going to investigate all of these listeners. I am, however, going to take a look at the CaretListener, since it can be used to monitor changes to the current position in the JTextArea portion of your application.

What does that require? Well, looking at the CaretListener Javadoc,[14] you see that only one method, caretUpdate(), needs to be implemented.
You can see that as you type into the input area, the label at the bottom of the application is updated to reflect the number of words and lines that exist. Figure 6-5 shows the sample output after pasting some text into the input area.

[14]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/CaretListener.html.

Figure 6-5.

SimpleEditor output

Listing 6-10 shows just how easy it is to use a TextArea and a CaretListener to create a very simple text editor.

Listing 6-10. SimpleEditor Class

```
class SimpleEditor( java.lang.Runnable ) :
def run( self ) :
frame = JFrame(

'Simple Editor',
layout = BorderLayout(),
defaultCloseOperation = JFrame.EXIT_ON_CLOSE

)
self.area = JTextArea(
rows = 8,
columns = 32,
caretUpdate = self.caretUpdate
)
frame.add( JScrollPane( self.area ), BorderLayout.CENTER )
self.words = JLabel( '# words: 0 # lines: 0' )
frame.add( self.words, BorderLayout.SOUTH )
frame.pack()
frame.setVisible( 1 )

def caretUpdate( self, event, regexp = None ) :
if not regexp :
regexp = re.compile( '\W+', re.MULTILINE )

pos = event.getDot()
text = self.area.getText()
if text.strip() == '' :

words = lines = 0
else :
words = len( re.split( regexp, text ) )
lines = len( text.splitlines() )
msg = '# words: %d # lines: %d' % ( words, lines )
self.words.setText( msg )
```

■ **Warning** this caretUpdate() method does not work well with large amounts of data. each time the caret (cursor) moves, the method is invoked, and it retrieves the entire text area and uses a regular expression to separate the data into "words." the splitlines() method then separates the data into "lines." this is very inefficient and suitable only for a trivial example such as this one.

Summary

This chapter is the first one to discuss the use of text input fields for your application scripts. It also introduced the important topic of threads, and the use of SwingWorker class instances to perform potentially long-running operations on a separate thread to keep the application from hanging. In the next chapter, you'll take a look at some other components that allow the users to provide input.

Chapter 7

Other Input Components

This chapter presents other input components available in the Swing class hierarchy. Specifically, it deals with some specialized text input fields, such as JPassword, three types of ComboBoxes (static, editable, and dynamic), and formatted text fields. Additionally, I combine some of these while discussing the JSpinner class at the end of the chapter.

Password Fields

One common reason for having a fairly small input field is to allow the users to enter a password. The good news is that the Swing class hierarchy includes an input component specifically for this purpose. Having recently seen the JTextField component, you shouldn't be too surprised to learn that a descendent component exists, called JPasswordField.[1] The bad news is that working with passwords, especially within a scripting language such as Jython, includes the possibility of security exposures, especially if the script writer doesn't think in terms of potential vulnerabilities.

Fortunately, the Swing developers did keep security in mind when designing Java as well as this input component.[2] For example, you may not realize it, but

unlike a normal text input field, the text in a JPasswordField can't be cut or copied. If you try to do so, a little bell sound is played to indicate that the requested action cannot be performed.

JPasswordField is a descendent of JTextField, which was discussed in Chapter 6. What happens when text is entered into JPasswordField? Figure 7-1 shows that as you enter text, each character is replaced by a userconfigurable echo character. Each instance of this component can, if you choose, have a unique echo character. To identify the character to be displayed for this input field, you use the setEchoChar() method.

Figure 7-1. *PasswordDemo sample output*

[1]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JPasswordField.html.

[2]Unfortunately, there are lots of ways for script writers to overlook good security practices, so be careful.

What does it take to use JPasswordField? Listing 7-1 contains the PasswordDemo class used to generate the output shown in Figure 7-1. There are some things I need to mention about this code though. Notice how, on line 13, the size keyword attribute is used to define the size of the frame to be displayed. This is especially useful in an example like this, which includes an initially empty JLabel component that will be used on the application window to display a message.

Listing 7-1. PasswordDemo Class

```
9|class PasswordDemo( java.lang.Runnable ) :
10| def run( self ) :
11| frame = JFrame(
12| 'PasswordDemo',
13| size = ( 215, 100 ),
14| layout = FlowLayout(),
15| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 16| )
```

```
17| frame.add( JLabel( 'Password:' ) )
18| self.pwd = frame.add(
19| JPasswordField(
20| 10,
21| actionPerformed = self.enter
22| )
23| )
24| self.msg = frame.add( JLabel() )
25| frame.setVisible( 1 )
26| def enter( self, event ) :
27| print 'ActionCommand: "%s"' % event.getActionCommand() 28| pwd =
self.pwd.getPassword()
29| if pwd == jarray.array( 'test', 'c' ) :
30| result = 'correct'
31| else :
32| result = 'wrong!'
33| self.msg.setText( 'Password is %s' % result ) .
```

If the frame size isn't specified, and a call to the frame.pack() method is used to determine the initial size of the application, space won't be allocated for the message field. This choice would require the ActionListener method (i.e., the enter() method on lines 26-33) to adjust the size of the application frame so the message text is visible. It's much easier to specify the frame size when the frame is constructed.

The JPasswordField instance, as shown on lines 19-22, uses the first parameter to specify the number of characters that are allowed. Notice that you don't have to do anything special to specify an echo character. If you don't like the default echo character, you can use the setEchoChar() method, the echoChar attribute, or the keyword constructor argument to specify the character to be displayed when the user enters data.

Is There an Echo in Here? Using the Character-Obfuscation Property

Sometimes it might be convenient to allow the users to "turn off" the character-obfuscation property of a password field. If this is the case, you can change the echoChar property of the field to have a value of zero (i.e., \x00 or chr(0)). When this is the case, any text in the field will be displayed "in the clear." To enable obfuscation, you simply specify a non-zero echoChar value. The

PasswordDemo2.py script file uses a button to demonstrate one way that this might be performed.

The important part of this script is the showHide() method, which acts as the event handler for the application button. Initially, this button has a value of Show and can be used to display the user input in the clear. Listing 7-2 shows that when the button is activated, the method uses the current text associated with the button to determine how to proceed. When the button text is Show, the password field echoChar attribute is set to zero and the text of the button changes to Hide. When the button text is Hide, the password field echoChar attribute is set to its original value and the text of the button is reset to Show.

Listing 7-2. showHide() Method from PasswordDemo2.py

```
def showHide( self, event ) :
button = event.getSource()
if button.getText() == 'Show' :

self.pwd.setEchoChar( chr( 0 ) ) button.setText( 'Hide' )
else :
self.pwd.setEchoChar( self.echoChar ) button.setText( 'Show' )
```

Figure 7-2 shows some sample images from the PasswordDemo2.py script. The first image shows what happens when text is entered. The next image shows the result of using the Show button. Notice how the button value has been changed and the password input field is now “in the clear.”[3] The next image shows the result of using the Submit button to verify the user input, and the final image shows how the Hide button can be used to obfuscate the password text field again.

Figure 7-2. *PasswordDemo2.py sample output*

[3]It is interesting to note that even though the password text is visible, you still can't copy its value to the system's clipboard.

The getPassword() Method

Reading the Javadoc for the JPassword class should help you understand that even though two getText() methods exist for the underlying text field, they have been deprecated and shouldn't be used. Each getText() method description includes the following statement:

■ **Note** For security reasons, this method is deprecated. Use the getPassword method instead.

Some interesting things should be noted about the differences between the getText() methods and the getPassword() method. One of the most obvious differences is that the return type of the getText() methods is String, whereas the return type of the getPassword() method is a character array. The primary reason for this is security. Strings are immutable and will stick around in memory until a garbage collection cycle can free the storage. A character array, on the other hand, can be replaced or overwritten immediately.

The event.getActionCommand() Method

What's so important about the event handler that it warrants a separate section? JPasswordField has JTextField as its base class. Because of this, it inherits some

properties from that class that could be possible security concerns. The following statement comes from the JTextField Javadoc page:

JTextField will use the command string set with the setActionCommand method if it’s not null; otherwise, it will use the text of the field as a compatibility with java.awt.TextField.

What does it mean for your application? It should be a warning that if you aren’t careful, the actionCommand attribute of the JPasswordField instance may be a password string, which might be another possible security exposure. So it is a best practice to provide a non-null actionCommand value for your password field instance. Listing 7-3 shows how easily this can be done using the actionCommand keyword attribute on the constructor call. When this is done, using the Enter key is indistinguishable from using the Submit button as far as the event handler is concerned.

Listing 7-3. Defining a Non-Null actionCommand Attribute Value self.pwd = frame.add(

```
JPasswordField(
10,
actionCommand = 'Submit', actionPerformed = self.enter

)
)
```

The JPasswordField Event Handler

You finally get to the ActionListener event handler method that is used to process the user-supplied password. You can see an example of this routine in lines 26-33 of Listing 7-1. As mentioned earlier, the data type returned by calling the getPassword() method is a Java array.

The simplest way to check the user-supplied password against the required password value is to create a Java array of the appropriate type and value and compare it with the value returned by the getPassword() method. To do so, you can use the jarray module. If you are unfamiliar with the array method in the jarray module,[4] it exports two functions for the creation of Java arrays for Jython scripts.[5] The exported functions are explained in Table 7-1.

Table 7-1. *The jarray Module Functions* **Function Signature**
array(sequence, typeCode) zeros(length, typeCode)

Description
Returns a Java array containing the values initialized using the specified sequence. Returns a Java array containing zero (or null) values of the specified length.

The possible values for the typeCode argument of these functions are listed in Table 7-2.

Table 7-2. *typeCode Values Used by the jarray Module* typeCode **Character Value Data Type of the Returned Array** 'b' byte
'c' char
'd' double
'f' float
'h' short
'i' int
'l' long
'z' Boolean

Using this information allows you to better understand the expression on line 29 in Listing 7-1. Listing 7-4 demonstrates this even more clearly. From this image, you can see just how easy it is to create a Java array using a string value to initialize the array.

Listing 7-4. Creating a Java array

```
wsadmin>import jarray
wsadmin>
wsadmin>jarray.array( 'test', 'c' ) array(['t', 'e', 's', 't'], char) wsadmin>
```

Using functions from the jarray module allows you to simplify the testing of the value returned by the JPasswordField getPassword() method call. So, you need to be able to compare the value entered by the users against the appropriate password value. Unfortunately, you also need to be able to view the user password in this simple application. Please don’t use this kind of technique—i.e., hard-coded passwords—in your applications.

[4]Seehttp://www.jython.org/javadoc/org/python/modules/jarray.html. [5]Table 7-2

only lists the function signatures that have a typeCode argument for simplicity's sake. Converting jarray Values to Strings

What if you need a Jython string in your script? Can't you use the toString() method to convert the array value to a string? Unfortunately, the data types returned by the jarray array() function and the JPasswordField getPassword() method are, in fact, of type org.python.core.PyArray. Listing 7-5 demonstrates this.

Listing 7-5. Java array type

```
wsadmin>import jarray
wsadmin>from javax.swing import JPasswordField wsadmin>
wsadmin>result = jarray.array( 'test', 'c' ) wsadmin>type( result )
<jclass org.python.core.PyArray at 1926329041> wsadmin>
wsadmin>pwd = JPasswordField( 'test' )
wsadmin>type( pwd.getPassword() )
<jclass org.python.core.PyArray at 1926329041> wsadmin>
```

So how do you convert this Java array of characters into a string that can be passed to one of the wsadmin scripting object methods? Listing 7-6 shows a couple of ways to do this. The first builds a simple array of the individual characters, and then uses the string join() method to return a string formed by concatenating the characters of the array with an empty string between each. The second shows how simple this operation can be when list comprehension is used instead.

Listing 7-6. Converting a Java array of characters to a string

```
wsadmin>result = []
wsadmin>for ch in pwd.getPassword() :
wsadmin> result.append( ch )
wsadmin>
wsadmin>str( result )
"['t', 'e', 's', 't']"
wsadmin>".join( result )
'test'
wsadmin>
wsadmin>
```

```
wsadmin>result = []
wsadmin>for ch in pwd.getPassword() :
wsadmin> result.append( ch )
wsadmin>
wsadmin>".join( result )
'test'
wsadmin>
wsadmin>result = ".join( [ ch for ch in pwd.getPassword() ] ) wsadmin>result
'test'
wsadmin>
```

Why do you need to worry about this? All you have to do is not set the ActionCommand string, and you will be able to retrieve the password as a string when the users press Enter, right? Not really. What happens when your application has multiple input fields, such as a user ID as well as a password? What about when you want to have two password fields for verification purposes? What happens when you also need or want to have a button? Then your script will have to use the JPasswordField getPassword() method to retrieve the Java array of characters. Then, before you can pass it to a WebSphere scripting object, you need some way to easily convert it from a Java array of characters into a Jython string. All this really means is that you need to know how to perform this type of conversion.

Choosing from a List

One nice technique for allowing users to provide input in your application is to provide a list of values and allow the users to select one. The Swing component that provides this kind of choice is the JComboBox. If you are interested in what that looks like, take a look at the sample output shown in Figure 7-3.

Figure 7-3. *ComboBoxDemo sample output*
There are some things that you should know about a JComboBox instance:

• The JComboBox class doesn't include an actionPerformed attribute that can be used as keyword argument in the constructor call. So you must either use the addActionListener() method call or the actionListener keyword argument to identify the ActionListener instance to be used.

• One of the side effects of this decision is that the application class has to be a descendent of the ActionListener class, and it must have an event handler method named actionPerformed(). An example of this can be seen on lines 9, 21, and 24 in Listing 7-7.

• It is important to realize that only one ComboBox item can be selected at a time. Multiple items cannot be selected using this component.

• The ActionListener event handler code can use the event.getSource() method to obtain a reference to the JComboBox instance. This technique allows you to access the ComboBox object easily, an example of which is shown on lines 25 and 26 in Listing 7-7. An alternative is to have the application keep an object instance reference to the JComboBox object.

Listing 7-7. ComboBox Class

```
9|class ComboBoxDemo( java.lang.Runnable, ActionListener ) : 10| def run( self
) :
11| frame = JFrame(
12| 'ComboBoxDemo',
13| size = ( 200, 100 ),
14| layout = FlowLayout(),
15| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 16| )
17| frame.add( JLabel( 'Pick one:' ) )
18| choices = 'The,quick,brown,fox,jumped'.split( ',' ) 19| choices.extend(
'over,the,lazy,spam'.split( ',' ) ) 20| ComboBox = frame.add( JComboBox(
choices ) ) 21| ComboBox.addActionListener( self )
22| self.msg = frame.add( JLabel() )
23| frame.setVisible( 1 )
24| def actionPerformed( self, event ) :
25| ComboBox = event.getSource()
```

26| msg = 'Selection: ' + ComboBox.getSelectedItem() 27| self.msg.setText(msg)

Editing a ComboBox

One of the questions that can come up when you are looking at the JComboBox class is whether it can be edited. The simple answer is yes. There is an editable attribute that you can set to allow users to enter a value that isn't on the list. Figure 7-4 shows some sample output of the EditableComboBox.py script.

Figure 7-4. *EditableComboBox sample output*

What do you need to do to allow users to enter off-list values? Interestingly enough, you only have to set the editable attribute on the JComboBox instance. Listing 7-8 shows just how simple it is to do so.

Listing 7-8. Making a ComboBox Editable

```
ComboBox = frame.add(
JComboBox(
self.choices,
editable = 1

)
)
```

There is one important thing to note, however. Even though the event handler method can obtain a user value that isn't in the original list, the list of items doesn't change. So, if the users want to use the same value again, they will have to re-enter it. This should make you wonder if there is a way to add and remove items from the list.

Using the DynamicComboBox

There is also a DynamicComboBox, but it requires a bit more effort and code to use it well. First, take a quick look at some sample output from this sample

application, and then you can take a look at and learn about the instructions required to generate the desired results.

Figure 7-5 shows how the list of items in the ComboBox can be removed and added to completely replace the list. What does it take to completely replace the items in the list? The next three listings show methods from the DynamicComboBox.py script.

Figure 7-5. *DynamicComboBox sample output*

Listing 7-9 shows the run() method, which creates the various Swing components, places them on the application, and assigns the appropriate ActionListener event handler for the JComboBox and the new Remove button. This is where you'll see the initial list of nine ComboBox items (i.e., lines 22 and 23).

Listing 7-9. DynamicComboBox run() Method

```
12|class DynamicComboBox( java.lang.Runnable, ActionListener ) :
13| def run( self ) :
14| self.frame = frame = JFrame(
15| 'DynamicComboBox',
```

```
16| size = ( 310, 137 ),
17| layout = BorderLayout(),
18| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
19| )
20| panel = JPanel()
21| panel.add( JLabel( 'Pick one:' ) )
22| self.choices = 'The,quick,brown,fox,jumped'.split( ',' )
23| self.choices.extend( 'over,the,lazy,spam'.split( ',' ) )
24| self.ComboBox = ComboBox = JComboBox(
25| self.choices,
26| editable = 1
27| )
28| ComboBox.addActionListener( self )
29| panel.add( ComboBox )
30| frame.add( panel, BorderLayout.NORTH )
31| panel = JPanel()
32| self.RemoveButton = JButton(
33| 'Remove',
34| actionPerformed = self.remove
35| )
36| panel.add( self.RemoveButton )
37| frame.add( panel, BorderLayout.CENTER )
38| panel = JPanel( alignmentX = Component.CENTER_ALIGNMENT )
39| self.msg = panel.add( JLabel( 'Make a selection' ) )
40| frame.add( panel, BorderLayout.SOUTH )
41| frame.setVisible( 1 ) .
```

One thing that might surprise you is the use of JPanel instances. The first is created on line 20. Then, you add a label (line 21) and a ComboBox (line 29) to it. Finally, this panel is added to the frame on line 30. This allows both the label and the ComboBox to be kept together as a group of components, and then this panel can be positioned on the top of the frame using the BorderLayout.NORTH constant, as shown on line 30.

The creation of a second panel, on line 31, may surprise you. If you didn’t do this and simply added the button to the frame using the BorderLayout.CENTER constant, the button would increase to fill the available space, thus making it too large. By placing the button in a panel, as on line 36, the panel can be sized to fill the available space, while leaving the button at its preferred size in the

middle of the panel.

Then you create yet another panel, on line 38. This time you use the alignmentX keyword assignment to configure the horizontal alignment accordingly. If you chose to simply add the message label (e.g., self.msg on line 39) to the BorderLayout.SOUTH position instead, the message label would be aligned to left and would look out of place.

The actionPerformed() method, shown in Listing 7-10, is the ActionListener event handler that is associated with the ComboBox by the statement on line 28 in Listing 7-9. This method is invoked when someone uses a mouse button or a keyboard event to make a selection on the ComboBox. It is especially interesting to note that the item returned by the call to the getSelectedItem() method, as shown on line 44, might in fact return a value that doesn't currently exist in the list of items currently associated with the ComboBox. This is only possible when the ComboBox editable attribute is true, as you can see on line 26 in Listing 7-9.

Listing 7-10. DynamicComboBox actionPerformed() Method

```
42| def actionPerformed( self, event ) :
43| cb = self.ComboBox
44| item = cb.getSelectedItem().strip()
45| items = [
46| cb.getItemAt( i )
47| for i in range( cb.getItemCount() )
48| ]
49| if item :
50| if item not in items :
51| cb.addItem( item )
52| self.RemoveButton.setEnabled( 1 )
53| msg = 'Selection: "%s"' % item
54| self.msg.setText( msg )
55| else :
56| cb.setSelectedIndex( 0 )
```

This event handler is responsible for determining if the specified item is valid, and whether or not it currently exists on the list. If it doesn't exist, a call is made to the addItem() method (line 51) to add it to the list of items currently associated with the ComboBox. Why do you call the setEnabled() method when

an item is added? Because it is possible for the remove method, shown in Listing 7-11, to reduce the number of items on the ComboBox to one. At this point, the Remove button should be disabled.

Listing 7-11. DynamicComboBox remove() Method

```
57| def remove( self, event ) :
58| cb = self.ComboBox
59| index = cb.getSelectedIndex()
60| item = cb.getSelectedItem()
61| try :
62| cb.removeItem( item )
63| self.msg.setText( 'Item removed: "%s"' % item )
64| except :
65| self.msg.setText( 'Remove request failed' )
66| self.RemoveButton.setEnabled( cb.getItemCount() > 1 )
```

The remove() method, shown in Listing 7-11, is associated with the button, as shown on line 34 in Listing 7-9. This method is invoked only when the user selects the Remove button, and the event handler routine is responsible for removing the current ComboBox item. It is also responsible for disabling the button when the number of items on the ComboBox list is reduced to one.[6]

[6]The expression (i.e., cb.getItemCount() > 1) will be true (i.e., 1) when more than one item is present on the list; this keeps the button enabled. When the list only has one item, the expression is false (i.e., 0) and the button is disabled.

Formatted Text Fields

There are times when it is important to control the way information is displayed by a text field. For situations such as this, Swing provides the JFormattedTextField class[7] and the Format class hierarchy.[8]

A formatted text field allows you to specify the kinds of values that are appropriate for a specific field and determine how these values are displayed. What kind of formatting can you apply? Well, you can format a text field as a number (e.g., as an integer, a floating point, or as currency). You can also specify a pattern to identify how a value should appear (e.g., a date, Social Security Number, or even a telephone number).

Take a quick look at a sample application that uses the various instances of formatted numbers.[9] Figure 7-6 shows the output of this simple application.

Figure 7-6. *FormattedTextFieldDemo sample output*

What does it take to generate the output? You might be surprised to see just how easy it is. Listing 7-12 contains the code from the FormattedTextFieldDemo.py script used to produce the output shown in Figure 7-6.[10]

Listing 7-12. FormattedTextFieldDemo Class

```
class FormattedTextFieldDemo( java.lang.Runnable ) :
def addFTF( self, name ) :
pane = self.frame.getContentPane()
pane.add( JLabel( name ) )
pane.add(

JFormattedTextField(
eval( 'NumberFormat.' + name ),
value = 12345.67890,
columns = 10

)

)
def run( self ) :
self.frame = frame = JFrame(
'FormattedTextFieldDemo',
layout = GridLayout( 0, 2 ),
defaultCloseOperation = JFrame.EXIT_ON_CLOSE
)
self.addFTF( 'getInstance()' ) self.addFTF( 'getCurrencyInstance()' )
self.addFTF( 'getIntegerInstance()' ) self.addFTF( 'getNumberInstance()' )
self.addFTF( 'getPercentInstance()' ) frame.pack()
frame.setVisible( 1 )
```

[7] Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/JFormattedTextField.htr
[8]Seehttp://docs.oracle.com/javase/8/docs/api/java/text/Format.html.
[9]Note: You'll revisit the JFormattedTextField class in Chapter 14.
[10]Note: The eval() function, as shown in line 15, is normally discouraged. However, using it in this example allowed the addFTF() method call to be greatly simplified and much shorter.

Looking closely at this script, you can see that the format of the value to be displayed in each field is defined by the NumberFormat instance that is used to instantiate each formatted text field.
The generated output may make you wonder about the kind of control that is provided by the NumberFormat class. Looking at the NumberFormat Javadoc,[11] you see that you can modify the attributes in Table 7-3. By the way, this table shows the initial/default settings for each of these attributes for the specified NumberFormat instance type.

Table 7-3. *NumberFormat Instance Type Attributes*
NumberFormat Instance Type Integer Part Fraction Part Minimum Digits

getInstance() 1
getCurrencyInstance() 1
getIntegerInstance() 1
getNumberInstance() 1
getPercentInstance() 1

Maximum Digits Minimum Digits Maximum Digits MAXINT 0 3
MAXINT 2 2
MAXINT 0 0
MAXINT 0 3
MAXINT 0 0

How are you supposed to understand this information? Well, take a look at a row in the table, e.g., the one for the currency instances. It differs from the others in that these kinds of values have a minimum and maximum of two fractional digits. That's why the value in the currency row of Figure 7-6 appears as $12,345.68. If you execute that script and enter a value of $0 in the input field, it will be reformatted as $0.00, which makes sense when you look at the row in Table 7-3 specifying currency values. There will be a minimum of one digit

before and exactly two digits after the decimal point.

One important point to note about these types of constructors for the NumberFormat class is that the instance attributes are specific to the locale of the operating system on which the wsadmin script is being executed. If your application requires a specific locale formatting, it might be specified as a parameter on the instance constructor:

format = NumberFormat.getCurrencyInstance(Locale.FRENCH) [11]See http://docs.oracle.com/javase/8/docs/api/java/text/NumberFormat.html.

Using a JSpinner Text Field

The last of the text input fields to be discussed here makes use of the javax.swing.JSpinner class. What does the JSpinner object look like? Each JSpinner field has a text input field and two small buttons. Figure 7-7 shows a trivial JSpinner field displaying the days of the week.

It should be obvious that the text field in this particular example isn't wide enough to display all of the characters in the longest weekdays. First, take a look at the class used to display this sample output. Listing 7-13 shows the Spinner1 class from this script file.[12]

Figure 7-7. *Sample JSpinner text field*

Listing 7-13. Spinner1 Class

```
from java.text import DateFormatSymbols as DFS
...
class Spinner1( java.lang.Runnable ) :

def run( self ) :
```

```
frame = JFrame(
'Spinner1',
layout = FlowLayout(),
defaultCloseOperation = JFrame.EXIT_ON_CLOSE

)
daysOfWeek = [ dow for dow in DFS().getWeekdays() if dow ]
frame.add( JSpinner( SpinnerListModel( daysOfWeek ) ) )
frame.pack()
frame.setVisible( 1 )
```

Some DateFormatSymbols Methods

To understand this example, you need to understand a few concepts. Let's begin with the getWeekdays() method of the DateFormatSymbols module.[13] Listing 7-14 shows an interactive wsadmin session showing what is returned by the DateFormatSymbols' getWeekdays() and getMonths() methods.[14]

Listing 7-14. DateFormatSymbols methods

[12] Line 5 is included to show how an alias for the DateFormatSymbols was defined. This allowed line 17 to be short enough to be fit easily in the available space.

[13]Seehttp://docs.oracle.com/javase/8/docs/api/java/text/DateFormatSymbols.html

[14]There are many more methods provided by the DateFormatSymbols class that aren't covered in this book. Feel free to experiment with and use these other methods.

```
wsadmin>from java.text import DateFormatSymbols as DFS
wsadmin>
wsadmin>DFS().getWeekdays()
array(['', 'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday'], java.lang. String)
wsadmin>
wsadmin>len( DFS().getWeekdays() )
8
wsadmin>DFS().getMonths()
wsadmin>array(['January', 'February', 'March', 'April', 'May', 'June', 'July',
'August', 'September', 'October', 'November', 'December', ''], java.lang.String)
```

It should be no surprise to you that the getWeekdays() and getMonths() methods return Java arrays. What might be a surprise to you—it certainly was to me—was the fact that the getWeekdays() method returns eight values and the getMonths() method returns 13 values.[15] That's why the Spinner1 class, as shown in Listing 7-13, uses list comprehension to process the results of calling the getWeekdays() method and returns the non-empty values that exist.

The JSpinner Class

Unlike the JComboBox, JSpinner instances do not display any kind of drop-down list of values. Only the current value is visible. The buttons on the field can be used to display the next or previous values. Why would you use a spinner instead of a ComboBox? Spinners are normally used when the number of valid items is too large to display. Does the use of the JSpinner class force the users to use the buttons? No, it doesn't.

Figure 7-8 shows the same Spinner1 application after various actions. You can see that the text portion of the spinner can be selected and user input (i.e., from the keyboard) can be used to filter or select the value to be displayed.

Figure 7-8. *Spinner value selections*

[15] This is *not* a bug, as is explained in the bug report found here: http://bugs.sun.com/bugdatabase/view_bug.do?bug_ id=4146173. It's a feature. Personally, I kind of like the explanation that exists in various places on the web that these eight-day weeks and 13-month years were created for managers and product planners to explain and justify the development schedules. But that's just a guess.

The various actions shown in Figure 7-9 are explained in Table 7-4.

Figure 7-9. *Default (numeric) spinner examples*

Table 7-4. *Spinner Value Selection, Explained*

Image Description

1 Shows the application output after the window is slightly widened.
2 Shows how the text in the input field can be selected.

3 After typing W (and clicking on the window edge to cause the text input field to be widened), you can see that the first value matching the specified text has been selected. It is interesting to note how the W has been deselected.

4 After selecting the entire text in the field again.
5 After typing T, notice that Tuesday has been selected, even though it precedes Wednesday in the list of values.
6 After typing h, notice how Thursday has been selected. This tells you that the value that you typed is used to match the selected portion of the text field.

This raises a number of questions in my mind. What kinds of values can be displayed in a spinner field? Do they have to be alphabetic strings? Let's take a look at the JSpinner Javadoc[16] to find out.

According to that page, there are only two JSpinner constructors—the default (i.e., empty) and one that uses something called a SpinnerModel argument. The default spinner constructor shows that it can be used to display numeric (i.e., integer). The images in Figure 7-9 show how numeric spinner values can be selected.

From left, the first image shows the initial, or default, value of 0. The second shows how a million is displayed. The third shows the maximum integer value (i.e., java.lang.Integer.MAX_VALUE of 2,147,483,647). The final image shows the value shown when the button is used to display the next value. Hopefully, you aren't too surprised by the fact that a wraparound occurs, and the value that is displayed is the smallest integer value
(i.e., java.lang.Integer.MIN_VALUE of -2,147,483,648).

One of the interesting things to note about this output is how the numeric values are formatted. From this output, you should quickly come to the realization that the text field portion of the spinner instance is a formatted text field.

The SpinnerModel Class

The SpinnerModel Javadoc[17] explains that this class is used for potentially unbounded values, which makes sense, especially looking back at the default spinner that you saw in Figure 7-9.

[16]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JSpinner.html.

[17]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/SpinnerModel.html.

One of the important points to note about the SpinnerModel is that it is an interface. Your applications can use one of your own classes based upon this interface class or one of the implementations provided by the Swing API, specifically SpinnerDateModel, SpinnerListModel, or SpinnerNumberModel. Looking back at Listing 7-13, you can see that the Spinner1 class uses the SpinnerListModel class. The constructor for this class was passed an array of strings. This particular example contains a short list of values, so a ComboBox may be a better choice in this type of situation. But it's your decision.

The SpinnerNumberModel class constructors have either zero or four parameters. The zero parameters constructor was used by the default JSpinner() constructor shown on line 15 of Listing 7-15.

Listing 7-15. Spinner2 Class

```
8|class Spinner2( java.lang.Runnable ) :
9| def run( self ) :
10| frame = JFrame(
11| 'Spinner2',

12| layout = FlowLayout(),
13| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
14| )
15| frame.add( JSpinner() )
16| frame.pack()
17| frame.setVisible( 1 )
```

This is the one that was used to display a selection of four billion values, as shown in Figure 7-9. The four-parameter variant of the SpinnerNumberModel class constructor identifies:

- The initial value to be displayed
- The minimum valid value

• The maximum valid value
• The stepSize, or increment value, to be used

The Spinner3 class that shows how this might be used is demonstrated in Listing 7-16.

Listing 7-16. Spinner3 Class

```
class Spinner3( java.lang.Runnable ) :
def run( self ) :
frame = JFrame(

'Spinner3',
layout = FlowLayout(),
defaultCloseOperation = JFrame.EXIT_ON_CLOSE

)
frame.add(
JSpinner(
SpinnerNumberModel(

0, # Initial value
-3141.59, # Minimum value
+3141.59, # Maximum value

3.14159 # stepSize
)

)
)
frame.pack()
frame.setVisible( 1 )
```

One of the interesting points to note about this application is that no value exists above the maximum or below the minimum specified values. This differs from the Spinner2 class, which uses a wraparound effect.

Figure 7-10 shows what the default SpinnerDateModel looks like, at least for my locale. The multiple images show that you can select different portions of the date in the text field and then use either the spinner buttons or the up and down keys on your keyboard to change the date.

Figure 7-10. *Default SpinnerDateModel examples*

For example, I set the current date to 3/1/00 and selected the day of the month part (i.e., the 1). Pressing the down key changes the date to 2/29/00 (i.e., Leap Day). How do you specify March 1, 2000 as the starting date for the spinner? Listing 7-17 shows one way to do this and is part of the Spinner5.py script.

Listing 7-17. Sample Use of SpinnerDateModel

```
frame.add(
JSpinner(

SpinnerDateModel(
Date( 2000, 2, 1 ), # zero origin month
None, # minimum
None, # maximum
Calendar.DAY_OF_MONTH # Ignored by GUI

)
)
)
```

One of the things to remember about this example is that even though a calendarField argument is specified, this does not automatically cause the specified field to be selected on the application window. You might be surprised to see the cursor located at the beginning or end of your input field, depending on your locale.

The JSpinner Editor

What if you don’t like the way that the value (e.g., Date) is shown in the spinner text field? To change it, you need to change the default spinner editor, which is based on the kind of SpinnerModel being used. Four spinner editors are provided by Swing:

• JSpinner.DateEditor—Used for SpinnerDateModel instances

• JSpinner.ListEditor—Used for SpinnerListModel instances
• JSpinner.NumberEditor—Used for SpinnerNumberModel instances
• JSpinner.DefaultEditor—A simple base class for more specialized editors

Figure 7-11 shows the sample output that is generated using a different date pattern used by the Spinner6 class shown in Listing 7-18.

Figure 7-11. *The Spinner6 output using a different date pattern*

Lines 27-30 show how you can specify a different date display pattern[18] for the formatted text field used by the spinner instance.

Listing 7-18. Spinner6 Class

```
10|from javax.swing import SpinnerDateModel
11|class Spinner6( java.lang.Runnable ) :
12| def run( self ) :
13| frame = JFrame(
14| 'Spinner6',
15| layout = FlowLayout(),
16| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
17| )
18| spinner = JSpinner(
19| SpinnerDateModel(
20| Date( 2000, 2, 1 ), # zero origin month
21| None, # minimum
22| None, # maximum
23| Calendar.DAY_OF_MONTH # Ignored by GUI
24| )
25| )
26| spinner.setEditor(
27| JSpinner.DateEditor(
28| spinner,
29| 'dd MMM yy'
30| )
31| )
32| frame.add( spinner )
33| frame.pack()
```

```
34| frame.setVisible( 1 )
```

This is just a simple example that should provide you with enough of a start for your applications.

Summary

This chapter explained many new input components that you can now utilize. By now, you should be getting a better feel for how Swing components can be used to provide user-friendly input choices for the users of your applications. In the next chapter, you will turn your attention to selectable input components.

[18] See http://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html.

Chapter 8

Selectable Input Components

So far, you have learned how to use simple buttons and a variety of text input fields. In this chapter, you'll take a look at using some other input components, ones that can be selected. All of the components in this chapter have two possible states, they are either selected or not. I'll start by describing toggle buttons, proceed to check boxes, and then to radio buttons. I will also discuss how to group these fields in order to make interesting user presentations.

Toggle Buttons

Simple push buttons are used to initiate an event of some kind. Occasionally, it can be useful for a button to have an associated state to convey additional information to the users. Toggle buttons perform this role. Thinking of this obvious difference makes me think of when I was little. The radio in my parents' car had buttons that you would press to select a station. It was quite obvious which of the buttons had been selected. You will learn how to group toggle buttons together so that only one button can be selected at a time.

Before you do that, though, you need to better understand toggle buttons and their states. The toggle button state can indicate whether the button has been selected/clicked or not. What does a toggle button look like? Figure 8-1 shows a simple application window containing a single toggle button, before and after it

is pressed. In order to accentuate the button's state, this application also changes the button's text based on its selected attribute.

Figure 8-1. *Sample toggle button application output*

What does it take to create a JToggleButton?[1] Listing 8-1 shows the ToggleButton class that instantiates a toggle button using the specified text and identifies the event handler to be invoked when the button state changes. The most significant difference between simple buttons and toggle buttons is the presence of the selected state. A push button, on the other hand, initiates an action when the button is clicked. Because of this, the event handler of the toggle button invokes the itemStateChanged() method of the ItemListener interface.[2] By comparison, the event handler for a JButton will invoke the actionPerformed() method of the ActionListener interface.[3] You can see this difference in the itemStateChanged keyword argument of the JToggleButton constructor in Listing 8-1.

[1] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JToggleButton.html.
[2]Seehttp://docs.oracle.com/javase/8/docs/api/java/awt/event/ItemListener.html.
[3]Seehttp://docs.oracle.com/javase/8/docs/api/java/awt/event/ActionListener.html.

Listing 8-1. ToggleButton Class

```
8|class ToggleButton( java.lang.Runnable ) :
9| def run( self ) :
10| frame = JFrame(

11| 'Toggle Button',
12| layout = FlowLayout(),
13| size = ( 275, 85 ),
14| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
15| )
16| button = JToggleButton( # Make a toggle button
17| 'Off' , # Initial button text
```

```
18| itemStateChanged = self.toggle # Event handler
19| )
20| frame.add( button )
21| frame.setVisible( 1 )
22| def toggle( self, event ) :
23| button = event.getItem()
24| button.setText( [ 'Off', 'On' ][ button.isSelected() ] )
```

The toggle method, shown in lines 22-24 of Listing 8-1, uses the getItem() method of the event that caused the method to be called in order to determine which component had a state change. It wasn't necessary to do this in this simple application, because there is only one component that has this property. However, this example is useful when you have multiple components that share an event handler. In fact, you'll see this scenario in the next example.

Check Boxes

While reviewing the Javadoc for the JCheckBox class,[4] I found it kind of interesting to see that it was, in fact, based on the JToggleButton class that you just read about. This makes sense because toggle buttons and check boxes both have a simple on/off state and react when clicked. Using this knowledge, you can easily create an application based on the ToggleButton code that has some check boxes. Figure 8-2 shows the some sample output of this application.

Figure 8-2. *Sample output for check boxes*

Listing 8-2 shows one way that this can be done.[5] A point of interest is how the event handler uses the getText() method (line 31) to retrieve the text of the check box that caused the state change method to be called. As mentioned earlier, this allows the check boxes to share the same event handler.

[4] Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/JCheckBox.html.

[5]Initially, I had the code to create and add each JCheckBox to the frame on a single line. Width limitations for these listings, however, meant that it would be best, at least for this example, to create and use the addCB() method in lines 9-

15 instead.

Listing 8-2. CheckBoxes Class

```
8|class CheckBoxes( java.lang.Runnable ) :
9| def addCB( self, pane, text ) :
10| pane.add(
11| JCheckBox(
12| text,
13| itemStateChanged = self.toggle
14| )
15| )
16| def run( self ) :
17| frame = JFrame(
18| 'Check Boxes',
19| layout = FlowLayout(),
20| size = ( 250, 100 ),
21| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 22| )
23| cp = frame.getContentPane()
24| self.addCB( cp, 'Yes' )
25| self.addCB( cp, 'No' )
26| self.addCB( cp, 'Maybe' )
27| self.label = frame.add( JLabel( 'Nothing selected' ) ) 28| frame.setVisible( 1 )
29| def toggle( self, event ) :
30| cb = event.getItem()
31| text = cb.getText()
32| state = [ 'No', 'Yes' ][ cb.isSelected() ]
33| self.label.setText( '%s selected? %s' % ( text, state ) )
```

■**Note** this example doesn’t use the selected parameter of the JCheckBox constructor, so each of the check boxes starts with the state as deselected.

Radio Buttons

One of the differences between check boxes and radio buttons is that radio buttons are most useful when they are grouped. This allows one, and only one, radio button in the group to be selected at a time. Take a look at the previous example again; this time, however, it uses radio buttons instead of check boxes.

Figure 8-3 shows sample output of this simple application, which is now using radio buttons instead of check boxes.

Figure 8-3. *Sample output for radio buttons*

Listing 8-3 shows the RadioButtons class that generates the output shown in Figure 8-3. What, if anything, do you notice about this class? Take a few moments to compare it to the CheckBoxes class in Listing 8-2. You should find very few differences between these two classes.

Listing 8-3. RadioButtons Class

```
9|class RadioButtons( java.lang.Runnable ) :
10| def addRB( self, pane, bg, text ) :
11| bg.add(
12| pane.add(
13| JRadioButton(
14| text,
15| itemStateChanged = self.toggle 16| )
17| )
18| )
19| def run( self ) :
20| frame = JFrame(
21| 'Radio Buttons',
22| layout = FlowLayout(),
23| size = ( 250, 100 ),
24| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 25| )
26| cp = frame.getContentPane()
27| bg = ButtonGroup()
28| self.addRB( cp, bg, 'Yes' )
29| self.addRB( cp, bg, 'No' )
30| self.addRB( cp, bg, 'Maybe' )
31| self.label = frame.add( JLabel( 'Nothing selected' ) ) 32| frame.setVisible( 1 )
33| def toggle( self, event ) :
34| text = event.getItem().getText()
35| self.label.setText( 'Selection: ' + text )
```

As mentioned earlier, you also need a ButtonGroup[6] to indicate the collection to which the new JRadioButton[7] object should be added. This ButtonGroup is created on line 27 and passed to the addRB() method on lines 28-30. So that shouldn't be too much of a surprise. Were you surprised that with one statement —in lines 11-18—you can:

- Create a radio button (line 13)
- Specify the associated text (line 14)
- Specify the ActionListener (i.e., itemStateChanged) event handler (line 15)
- Add the ActionListener to the specified pane (line 12)
- Add the ActionListener to the specified button group (line 11)

I found this quite nice, and I appreciate how easy it is to understand the code required to do this. Granted, the example isn't perfect, but it is very easy to replace an application component with a closely related one.

[6]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/ButtonGroup.html.

[7]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JRadioButton.html.

Toggle Buttons in a Button Group

In the first part of this chapter I mentioned how the radio buttons of my parents' car were similar to toggle buttons. Thinking of this made me wonder how difficult it would be to simulate this feature in a script. All you need to do is place a bunch of toggle buttons in a button group, right? Well, that's pretty much the case. There are a few extra things that you need to do (initialize one of the buttons as selected and display the correct message), but that's pretty much it. Figure 8-4 shows the sample output for this application.

Figure 8-4. *A group of toggle buttons*

What does it take to do this? Not much. In fact I think that you'll agree that Listing 8-4 looks very similar to Listing 8-3, where you used a button group to identify a group of radio buttons.

Listing 8-4. ButtonGroupDemo

```
class ButtonGroupDemo( java.lang.Runnable ) :
def addRB( self, pane, bg, text ) :
bg.add(

pane.add(
JToggleButton(
text,

selected = ( text == '1' ),
itemStateChanged = self.toggle
)
)

)
def run( self ) :
frame = JFrame(
'ToggleButton Group',
layout = FlowLayout(),
size = ( 265, 100 ),
defaultCloseOperation = JFrame.EXIT_ON_CLOSE
)
cp = frame.getContentPane()
bg = ButtonGroup()
for i in range( 1, 6 ) :
self.addRB( cp, bg, `i` )
self.label = frame.add( JLabel( 'Selection: 1' ) )
frame.setVisible( 1 )
def toggle( self, event ) :
text = event.getItem().getText()
self.label.setText( 'Selection: ' + text )
```

Here’s a question for you. Do toggle buttons in a button group act any differently than those that aren’t in a button group? The answer is yes they do. Once a toggle button within a button group is selected, it can’t be deselected. In fact, this is the same behavior shown when radio buttons are used. Initially, you can have all of the radio buttons (or toggle buttons) in a button group deselected, but once one has been selected, one will always be selected.[8]

Summary

This chapter, even though it is fairly short, discusses and describes selectable input components. One of the interesting things that I found while investigating these components was the fact that all the selectable classes are based on the JToggleButton class. This is what led me to investigate using multiple toggle buttons within a button group. I hope that you find these an interesting addition to your collection of user interface components.

[8]Unless, of course, you have something like an event handler deselect everything in the group. **Chapter 9**

Providing Choices, Making Lists

You are likely to encounter situations where it would be nice to provide your users with a list of choices. For example, you've probably selected the name of the city where you live from a list. Maybe you want to build an application to keep track of the books or movies that you own. Fortunately, Swing provides the JList[1] component, which allows programmers to build and display lists of this sort. In this chapter, you learn how to build and display a list of items. You will also learn how to manipulate a list in the event handler method associated with another component, such as a button.

Making a List and Checking It Twice

How hard is it to make a list? Not very. Listing 9-1 shows how little code is required to create a list using a group of words from a string. Another version of this script, called List1a.py, is provided that shows you how to build a JList using a java.util.Vector.

Listing 9-1. The List1 Class class List1(java.lang.Runnable) :
def run(self) :
frame = JFrame(

'List1',
size = (250, 200),
defaultCloseOperation = JFrame.EXIT_ON_CLOSE

)
data = 'Now is the time for all good spam'.split(' ') frame.add(JList(data))
frame.setVisible(1)

What can you do with a default JList? Figure 9-1 shows that you can select one or more items on this list with no additional code; you simply need to press the Ctrl key while additional items are selected.

[1]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JList.html.

Figure 9-1. *List1's sample output*

Wow, that was easy, right? You're all done and can move on to the next topic, right? Not quite. There are lots of things that you need to consider when your applications deal with lists. For example, the last image shows one problem that occurs when the size of the frame isn't large enough to show all of the entries in the list. There is no indication that additional items exist. Even worse, if you use the cursor down arrow to move the selection down the list, you can select items that are not visible to the user. What can you do about that?

Optional Scroll Bars

If your list has a limited (small) number of items (such as the days of the week),

you might want to use something like a JComboBox[2] instead of a JList. Frequently, however, the number of items on the list won't easily fit in the application window. When this happens, you only need to wrap the JList instance within a JScrollPane.[3] This is so easy to do that it is hard to imagine why you wouldn't want to always put your JList instance in a scroll pane. Listing 9-2 shows just how easy this can be. Compare line 15 to line 14 in Listing 9-1.

Listing 9-2. Wrapping a JList in a Scroll Pane

```
7|class List2( java.lang.Runnable ) :
8| def run( self ) :
9| frame = JFrame(
10| 'List2',
```

[2]As discussed in section 7.2.
[3]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JScrollPane.html.

```
11| size = ( 250, 100 ),
12| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
13| )
14| data = 'Now is the time for all good spam'.split( ' ' )
15| frame.add( JScrollPane( JList( data ) ) )
16| frame.setVisible( 1 )
```

What does this do to the application? Well, if the list contains more items than can be displayed in the available area, vertical and/or horizontal scroll bars will be added, as needed, to allow the users to determine which parts of the available information they want to view. Figure 9-2 shows the output of the List2 application. It has space for fewer lines and therefore requires a vertical scrollbar.[4] Once the size of the frame is large enough to display the complete list, the vertical scrollbar automatically disappears.

Figure 9-2. *List2's sample output*

The ScrollPane Viewport

The JScrollPane Javadoc includes a diagram that shows the relationship between the ScrollPane instance and the child component contained in it. One of the important concepts to note is that a JViewport[5] is created to determine the portion of the child component to be displayed.

■ **Tip** if you are interested in learning more about scroll panes, viewports, and scroll bars, i encourage you to take a look at the "how to Use scroll panes"[6] portion of the Java swing tutorials. this book doesn't delve into more detail about viewports, but they do show up in Chapter 12, where tables are discussed in detail.

[4] In case you are wondering, I simply held the Ctrl key and selected the odd list entries to produce these images.

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JViewport.html.

[6]See http://docs.oracle.com/javase/tutorial/uiswing/components/scrollpane.html.

Manipulating the List

You will frequently need your applications to manipulate a list in some way. For example, you may want to add, remove, or replace items on a list. To do this, you need to use the interface provided by the list model. What's a list model? It is the component that actually contains the list contents and provides methods that allow your applications to manipulate the list. If one is not passed to the JList constructor, a DefaultListModel[7] will be created to for you.[8] You previously saw this separation of the component and its data in Chapter 7, which discussed the SpinnerModel class.

Counting List Items

Let's take a quick look at how a list model can be used to count the number of times a specific value occurs on the list. Figure 9-3 shows some sample output for the List3 application. The first image (from the left) shows the initial look of the application. The second image shows what the application looks like after you enter some text into the input field (i.e., "the") and click the Count button. Note how the input field is cleared and the message portion of the application is updated to reflect the number of occurrences of the specified word.

Figure 9-3. *List3's sample output*

Listings 9-3 and 9-4 show the List3 class produces the output shown in Figure 9-3. Lines 21-25 show how the list of words is created, again using a simple array of strings. Lines 26-32 show how the list is wrapped in a scroll pane and added to the top of the application window using the BorderLayout.NORTH position constant.

Listing 9-3. List3's run() Method

```
13|class List3( java.lang.Runnable ) :
```

```
14| def run( self ) :
15| frame = JFrame(
16| 'List3',
17| size = ( 200, 200 ),
18| layout = BorderLayout(),
19| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
20| )
```

[7] See http://docs.oracle.com/javase/8/docs/api/javax/swing/DefaultListModel.html.
[8]This book discusses the use of the DefaultListModel, not the AbstractListModel, the ListModel, or the more interesting SortedListModel class discussed at http://java.sun.com/developer/technicalArticles/J2SE/Desktop/sorted_jlist/

```
21| data = (
22| 'Now is the time for all good spam ' +
23| 'to come to the aid of their eggs'
24| ).split( ' ' )
25| self.info = JList( data )
26| frame.add(
27| JScrollPane(
28| self.info,
29| preferredSize = ( 200, 110 )
30| ),
31| BorderLayout.NORTH
32| )
33| panel = JPanel( layout = GridLayout( 0, 2 ) )
34| panel.add(
35| JButton(
36| 'Count',
37| actionPerformed = self.count
38| )
39| )
40| self.text = panel.add( JTextField( 10 ) )
41| frame.add( panel, BorderLayout.CENTER )
42| self.msg = JLabel( 'Occurance count' )
43| frame.add( self.msg, BorderLayout.SOUTH )
44| frame.setVisible( 1 )
```

The middle portion of the application, containing the Count button and text input fields, is created and positioned on the application window in lines 33-41. Finally, the status message (a label field) is created and placed on the bottom of the application window in lines 42 and 43.

Listing 9-4 shows the count() method, which is identified as the button's actionListener event handler, on line 37 in Listing 9-3, when the button is created. It is here that you need to use the list model to count the number of times a specified value occurs in the associated list.[9]

Listing 9-4. List3's count() Method

```
45| def count( self, event ) :
46| word = self.text.getText()
47| model = self.info.getModel()
48| occurs = 0
49| for index in range( model.getSize() ) :
50| if model.getElementAt( index ) == word :
51| occurs += 1
52| self.msg.setText(
53| '"%s" occurs %d time(s)' %
54| ( word, occurs )
55| )
56| self.text.setText( " )
```

[9]You wouldn't want to use this technique on an unbounded list. And, if your list was large, you would want the list model processing to be performed on a separate thread (i.e., by a SwingWorker class instance).

Limiting the Selectable Items

Other list-manipulation actions include adding and removing items to and from the list. Before you take a look at what these operations require, think about it for a moment. When you want to add an item to the list, where do you want to add it? Should it be added to the beginning of the list, at the end, or somewhere in the middle? You also might need to consider if multiple occurrences of a value should be allowed on the list.

Do you want to allow multiple items to be selected? This might make sense if you want to allow the users to remove multiple items with a single button click,

but this might not always make sense. For example, if you want to allow the users to select an item on the list, and then allow a new item to be added, either before or after the selected item, then it doesn't make sense to allow multiple items to be selected at one time.

Earlier, you saw that the default list properties allow you to select multiple list items. How do you disable this? The JList class includes a selection mode attribute that can define the number of list items that can be selected. Table 9-1 describes the list selection mode values.

Table 9-1. *List Selection Mode Examples*

Selection Mode Description

SINGLE_SELECTION Either zero or one item may be selected.

SINGLE_INTERVAL_SELECTION Only one contiguous group of items may be selected.

MULTIPLE_INTERVAL_SELECTION The default mode allows multiple groups of items to be selected.

In Java applications, you would normally use the JList setSelectionMode() method to identify the mode to be used. In Jython, you can use this method call or you can use the selectionMode keyword argument when you create the JList object. Listing 9-5 shows an example of how easily this can be done using the constructor keyword argument

Listing 9-5. Using the List Selection Mode Keyword Argument

```
|26|        self.info = JList(                                          |
|27|            data,                                                   |
|28|            selectionMode = ListSelectionModel.SINGLE_SELECTION     |
|29|        )                                                            |
```

Reacting to List-Selection Events

Can you think of a reason that your application needs to know when an item on a list has been selected? What kinds of things might you want to do with the selected item? You might want to remove it, modify it, or insert a new item before or after the selected item. You might also want to initiate some more complex operation based on the user selection. It's all up to you.

If no item is selected, what kinds of things might you want to be able to do? What about adding items to the list? Where might you be able to add an item if you have to reference it (a selected item)? What about adding a new item first or last on the list? Those options make sense, don't they?

The reason I'm asking these questions is to help you think about things that you might want to consider as you are creating your own applications.
Do list selection events exist to be monitored? Sure. The challenge is trying to figure out what you want to do when a list selection event occurs. What do you want to happen when the users select a list item? It depends on what your application looks like and what it does.
What if you partition the application output to have the scrollable list and a text input field on top and some buttons below? Figure 9-4 shows one way to do this.

Figure 9-4. *List4's sample output*

Listing 9-6 shows how easily this can be done using Jython. To partition the application window, the frame uses a BorderLayout, with the scrollable pane containing the list instance positioned using the BorderLayout.NORTH constant (line 35), and another pane that uses a GridLayout to position the text field and buttons within it (lines 37-43). Then, this pane is positioned on the application frame using the BorderLayout.SOUTH constant (line 44).

Listing 9-6. List4's Class

```
14|class List4( java.lang.Runnable ) :
15| def run( self ) :
16| frame = JFrame(
17| 'List4',
18| size = ( 200, 222 ),
19| layout = BorderLayout(),
20| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
21| )
22| data = (
23| 'Now is the time for all good spam ' +
24| 'to come to the aid of their eggs'
25| ).split( ' ' )
```

```
26| self.info = JList(
27| data,
28| selectionMode = ListSelectionModel.SINGLE_SELECTION
29| )
30| frame.add(
31| JScrollPane(
32| self.info,
33| preferredSize = ( 200, 100 )
34| ),
35| BorderLayout.NORTH
36| )
37| panel = JPanel( layout = GridLayout( 0, 2 ) )
38| self.text = panel.add( JTextField( 10 ) )
39| panel.add( self.button( 'First' ) )
40| panel.add( self.button( 'Last' ) )
41| panel.add( self.button( 'Before' ) )
42| panel.add( self.button( 'After' ) )
43| panel.add( self.button( 'Remove' ) )
44| frame.add( panel, BorderLayout.SOUTH )
45| frame.setVisible( 1 )
46| def button( self, text ) :
47| return JButton( text, actionPerformed = self.insert )
48| def insert( self, event ) :
49| todo = event.getActionCommand()
50| word = self.text.getText()
51| print '%s: "%s"' % ( todo, word )
```

Now that you have an idea how the application will look, you might be able to figure out what you want to happen when a list item is selected. You can also think about changing the appearance of the application if you don't like the way it looks. What if you had the buttons and text field on the left and the list of items on the right? What would that look like? It takes very little time and effort to figure this out. Figure 9-5 shows the result of making these changes.

Figure 9-5. *List5's sample output*

Looking at Listing 9-7, you can see how few changes were needed to produce this output. Additionally, you now have an event handler that is called when a list selection operation has occurred (lines 35 and 51-54).
This method currently only produces output on the console, but it provides you with information about when a selection event occurs, as well as informs you that you can use the available information to determine whether a list item has been selected. You can do this by looking at the Javadoc for the ListSelectionListener class[10] and verifying that you only need to identify the valueChanged() method (see line 35).

Listing 9-7. List5's Class Reacting to List Selection Events

```
13|class List5( java.lang.Runnable ) :
14| def run( self ) :
15| frame = JFrame(
16| 'List5',
17| size = ( 200, 220 ),
18| layout = GridLayout( 1, 2 ),
19| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
20| )
21| panel = JPanel( layout = GridLayout( 0, 1 ) )
22| panel.add( self.button( 'Remove' ) )
23| panel.add( self.button( 'First' ) )
```

```
24| panel.add( self.button( 'Last' ) )
25| panel.add( self.button( 'Before' ) )
26| panel.add( self.button( 'After' ) )
27| self.text = panel.add( JTextField( 10 ) )
28| frame.add( panel )
29| data = (
30| 'Now is the time for all good spam ' +
31| 'to come to the aid of their eggs'
32| ).split( ' ' )
33| self.info = JList(
34| data,
35| valueChanged = self.selection,
36| selectionMode = ListSelectionModel.SINGLE_SELECTION 37| )
38| frame.add(
39| JScrollPane(
40| self.info,
41| preferredSize = ( 200, 100 )
42| )
43| )
44| frame.setVisible( 1 )
45| def button( self, text ) :
46| return JButton( text, actionPerformed = self.insert ) 47| def insert( self, event
) :
48| todo = event.getActionCommand()
49| word = self.text.getText()
50| print '%s: "%s"' % ( todo, word )
51| def selection( self, e ) :
52| index = e.getSource().getSelectedIndex()
53| if not e.getValueIsAdjusting() :
54| print 'selected %d' % index
```

[10]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/event/ListSelectionLis

Reacting to User (Text) Input

All right, you've given some thought as to how you want the application to look and decided that you want to do something when a list item is selected. What do you want to happen next? Wouldn't it be neat to have the application's buttons be enabled or disabled based on the user input? For example, it makes sense to enable the Remove button when a list item is selected. What about the other

buttons? When do you want them to be enabled? What if you only enable the buttons related to list insertion when the user specifies some text in the text input field?

The InputMethodListener class[11] looks easy to use, right? But will it solve the problem at hand? Unfortunately, it is not going to work in this scenario. The List6.py script contains an attempt to add an InputMethodListener event handler.[12] Unfortunately, when you try to execute this script you get an AttributeError exception. Why? Because the addInputMethodListener() method is provided by the javax.swing.text.JTextComponent class, not the Runnable or InputMethodListener classes.

Listing 9-8. List6.py Script Interesting Lines

```
15|from java.awt.event import InputMethodListener
16|class List6( java.lang.Runnable, InputMethodListener ) : | ...
29| self.addInputMethodListener( self )
| ...
69| def caretPositionChanged( self, e ) :
70| print 'caretPositionChanged() :', e
71| def inputMethodTextChanged( self, e ) :
72| print 'inputMethodTextChanged() :', e
```

Listing 9-8 shows how easy it is to make these kinds of changes. Why do I encourage you to look at failures? Well, there are a few very important reasons to do so:

[11] See http://docs.oracle.com/javase/8/docs/api/java/awt/event/InputMethodListener.htm

[12] Please note that the line in List6.py that corresponds to line 29 in Listing 9-8 is commented out. To see the exception you need to remove the “#” in column one of that statement before executing the script.

- To try new things
- To learn how to fail gracefully
- To learn from your mistakes

It only takes a few moments to search for AWT and Swing methods related to events that might be called when the users enter text. It only required you to insert or modify seven lines of code. Was this a huge investment of time and effort? No, and what you learned was priceless.

Is there another way to use an InputMethodListener? Certainly, you can create a class to implement the InputMethodListener interface and see if that works. The List6a.py script contains this attempt. Listing 9-9 shows the modified lines from List6.py to demonstrate this iteration.

Listing 9-9. List6a.py Script, Unique Lines Only

```
15|from java.awt.event import InputMethodListener
16|class IML( InputMethodListener ) :
17| def caretPositionChanged( self, e ) :
18| print 'caretPositionChanged() :', e
19| def inputMethodTextChanged( self, e ) :
20| print 'inputMethodTextChanged() :', e
21|class List6a( java.lang.Runnable ) :
| ...
33| self.text = panel.add( JTextField( 10 ) )
34| self.text.addInputMethodListener( IML() )
```

Unfortunately, if you test this script, you will see that modifying the input field does not generate any output. Additional research shows that another approach might be more viable. For this iteration, you see what you need to use a DocumentListener instead of an InputMethodListener. This example is found in List6b.py; the modified lines are shown in Listing 9-10.

Listing 9-10. List6b.py Script, Unique Lines Only

```
15|from javax.swing.event import DocumentListener
16|class DL( DocumentListener ) :
17| def changedUpdate( self, e ) :
18| print 'changedUpdate() :', e
19| def insertUpdate( self, e ) :
20| print 'insertUpdate() :', e
21| def removeUpdate( self, e ) :
22| print 'removeUpdate() :', e
23|class List6b( java.lang.Runnable ) :
| ...
35| self.text = panel.add( JTextField( 10 ) )
36| self.text.getDocument().addDocumentListener( DL() )
```

It is interesting to note that in order to have a DocumentListener for the input field, you can use the

getDocument() method of the JTextField class to identify the actual Document object that is associated with this input field. If you test this iteration, as shown in Figure 9-6, you can see that events are being generated and the corresponding DocumentListener methods are being called. The particular events were the result of typing a character and then using the Backspace key to delete it from the input field.

This approach might not be the best way to monitor changes to an input field, especially with something simple like the JTextField, which is used in these examples. Fortunately, there is another approach that you can investigate.

```
insertUpdate() : [javax.swing.text.GapContent$InsertUndo@...]
removeUpdate() : [javax.swing.text.GapContent$RemoveUndo@...]
```

Figure 9-6. *List6b's DocumentListener sample output*

For this example, you are going to look at what happens when you add a listener to the input field for KeyListener events.[13]

```
text: "b"
text: "ba"
text: "bac"
text: "baco"
text: "bacon"
```

Figure 9-7 shows that you can monitor the text input field using a KeyListener keyReleased() method call. Listings 9-11 and 9-12 show the List7 class used to produce this output.

```
text: "b"
text: "ba"
text: "bac"
text: "baco"
text: "bacon"
```

Figure 9-7. *List7's sample output*

Listing 9-11. List7's run() Method

```
class List7( java.lang.Runnable, KeyListener ) :
 def run( self ) :
 frame = JFrame(
 'List7',
 size = ( 200, 220 ),
```

```
21| layout = GridLayout( 1, 2 ),
22| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
23| )
24| panel = JPanel( layout = GridLayout( 0, 1 ) ) 25| self.buttons = {}
26| for name in 'First,Last,Before,After,Remove'.split( ',' ) : 27| self.buttons[
name ] = panel.add( self.button( name ) ) 28| self.text = panel.add(
29| JTextField(
30| 10,
31| keyReleased = self.typed
32| )
33| )
34| frame.add( panel )
35| data = (
36| 'Now is the time for all good spam ' +
37| 'to come to the aid of their eggs'
38| ).split( ' ' )
39| model = DefaultListModel()
40| for word in data :
41| model.addElement( word )
42| self.info = JList(
43| model,
44| valueChanged = self.selection,
45| selectionMode = ListSelectionModel.SINGLE_SELECTION 46| )
47| frame.add(
48| JScrollPane(
49| self.info,
50| preferredSize = ( 200, 100 )
51| )
52| )
53| frame.setVisible( 1 )
```

[13]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/KeyListener.html.

Listing 9-11 shows the run() method and Listing 9-12 shows the rest of the methods from this class.

Listing 9-12. The Remaining List7 Class Methods

```
54| def button( self, text ) :
55| return JButton(
56| text,
```

```
57| enabled = 0,
58| actionPerformed = self.doit
59| )
60| def doit( self, event ) :
61| todo = event.getActionCommand()
62| word = self.text.getText().strip()
63| List = self.info
64| pos = List.getSelectedIndex()
65| print '%s: "%s" pos: %d' % ( todo, word, pos )
66| if todo == 'Remove' :
67| List.getModel().remove( pos )
68| self.buttons[ todo ].setEnabled( 0 )
69| def selection( self, e ) :
70| if e.getValueIsAdjusting() :
71| si = e.getSource().getSelectedIndex() 72| self.buttons[ 'Remove' ].setEnabled(
si > -1 ) 73| def typed( self, e ) :
74| print 'text: "%s"' % self.text.getText().strip()
```

Figure 9-8 shows the progress in the application output using the next iteration of the script. This one is found in the List8.py script file. You can see which buttons are enabled before any text is entered, after some text is entered, after a single item is selected, and finally after a button is used to insert the new word on the list.

Figure 9-8. *List8's sample output*

You can look at the iterations of the application and see how small changes have been made with each. Finally, you get to the final iteration, which is found in the List10.py script file. This version includes the small improvement of putting the removed item text in the text input field. This allows you to do:

1. Select a the first item.
2. Click the Remove button (thus putting the item value into text field).
3. Click the Last button to move the first item to the end of the list.

Listing 9-13 contains the List10 class methods that allow your application to do these things. The run() method is not included because it’s almost identical to the run() method found in Listing 9-11. The List10 class does, however, include some interesting properties. You might not agree that these are advantages. Specifically, List10 takes advantage of the named keyword arguments to use the textCheck() method as:

• A KeyListener keyReleased() method
• A ListSelectionListener valueChanged() method
• A simple textCheck() class

Java purists may take offense at this flexibility. However, I think that most Jython programmers can see the value provided by the use of keyword argument lists in this type of application.

Listing 9-13. Remaining List10 Methods

```
57| def doit( self, event ) :
58| todo = event.getActionCommand()
59| word = self.text.getText().strip()
60| List = self.List
61| pos = List.getSelectedIndex()
62| model = List.getModel()
63| if todo == 'Remove' :
64| self.text.setText( List.getSelectedValue() )
65| model.remove( pos )
66| else :
67| if todo == 'First' :
68| model.insertElementAt( word, 0 )
69| elif todo == 'Last' :
70| model.insertElementAt( word, model.getSize() )
71| elif todo == 'Before' :
72| model.insertElementAt( word, pos )
73| else :
74| model.insertElementAt( word, pos + 1 )
75| self.text.setText( '' )
```

```
76| self.textCheck()
77| def textCheck( self, e = None ) :
78| word = self.text.getText().strip()
79| index = self.List.getSelectedIndex()
80| for name in 'First,Last'.split( ',' ) :
81| self.buttons[ name ].setEnabled( len( word ) > 0 )
82| for name in 'Before,After'.split( ',' ) :
83| self.buttons[ name ].setEnabled(
84| len( word ) > 0 and index > -1
85| )
86| self.buttons[ 'Remove' ].setEnabled( index > -1 )
```

One of the most impressive things about this example is that it demonstrates how easy it is to quickly produce applications. Being able to produce a rapid prototype is enormously valuable to programmers because they can then easily manipulate and interact with the application and decide what they like and dislike about the way it looks, as well as the way it responds to user input.

Summary

This chapter covered how to display and manipulate lists of items with an emphasis on iterating the application to test various representations of components on the frame as well as using a variety of listeners to improve your understanding of how each might or might not enhance the usability of the application. You saw just how quickly and easily a script can be modified to test a new approach so that you can discard those that don’t fit your specific needs.

Note: One point that this chapter didn’t cover was the fact that once modifications have been made to the list of values, the application should retrieve the current list’s contents using the ListModel methods. **Chapter 10**

Menus and MenuItems

Menus are one of the most common input mechanisms that applications use. When was the last time that you worked with a graphical application that didn’t include some sort of menu structure and hierarchy?

The main reason for creating and using menus in your applications is that it conveys to the users some of the actions that they can perform with the

application. So this is all about setting expectations and communicating at least some of the things that the applications can do.

This chapter shows how easily you can add a menu to your applications and how to make your applications react when the user selects a menu item.

The JMenu Class Hierarchy

You may not realize it, but menus are a kind of button. This makes a lot of sense when you think about what happens when you click a button and when you click on a menu item. Each causes some kind of event that needs an event handler to perform the desired action. There are differences, though. If there weren't, you wouldn't need another class. Take a moment to think about it. How would you describe a menu? I think that you'll agree that they are almost always a collection of individual words that convey information to the users. Each word or menu entry can be selected and then displays a related sub-menu or performs some kind of response. A button, on the other hand, is almost always used to initiate some kind of action or elicit a specific response.

You can see how menus and buttons are similar by looking at the JMenu class[1] Javadoc or by using the classInfo routine first mentioned in Chapter 4. The classInfo hierarchy for the JMenu class is shown in Listing 10-1.

Listing 10-1. JMenu Class Hierarchy

```
wsadmin>classInfo( JMenu )
javax.swing.JMenu
| javax.swing.JMenuItem
| | javax.swing.AbstractButton
| | | javax.swing.JComponent
| | | | java.awt.Container
| | | | | java.awt.Component
| | | | | | java.lang.Object
| | | | | | java.awt.image.ImageObserver | | | | | | java.awt.MenuContainer | | | | | |
java.io.Serializable | | | | java.io.Serializable
| | | java.awt.ItemSelectable
| | | javax.swing.SwingConstants | | javax.accessibility.Accessible | |
javax.swing.MenuElement
| javax.accessibility.Accessible | javax.swing.MenuElement
wsadmin>
```

[1]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JMenu.html.

Unlike regular buttons, or even toggle buttons, the result of activating a menu is to cause the associated list of items to be displayed. So how do you add a JMenu to the application? Since you are using a JFrame for almost all of your applications, let's take another quick look at the JFrame hierarchy, with a focus on the methods that contain the text "menubar"

Listing 10-2 shows this hierarchy. One important point to note is that the javax.swing.JFrame class has JMenuBar getters and setters, and the java.awt.Frame class has MenuBar getters and setters.[2]

Listing 10-2. JFrame MenuBar Methods

```
wsadmin>classInfo( JFrame, meth = 'menubar' ) javax.swing.JFrame
getJMenuBar, setJMenuBar

| java.awt.Frame
> getMenuBar, setMenuBar
| | java.awt.Window
| | | java.awt.Container
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver | | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | javax.accessibility.Accessible
| | java.awt.MenuContainer
| javax.swing.WindowConstants
| javax.accessibility.Accessible
| javax.swing.RootPaneContainer
wsadmin>
```

What does this mean? For one, it means you need to be careful. Listing 10-3 shows what I'm talking about. Since there are getters and setters for JMenuBars as well as MenuBars, you need to be certain that you remember to include the *J*, or you'll get an exception like the one shown in lines 12 and 13.

Listing 10-3. Adding a MenuBar to the JFrame

```
1|wsadmin>from javax.swing import JFrame, JMenuBar, JMenu
2|wsadmin>
```

```
3|wsadmin>f = JFrame( 'Frame Title' )
4|wsadmin>b = JMenuBar()
5|wsadmin>m = b.add( JMenu( 'Help' ) )
6|wsadmin>f.setMenuBar( m )
7|WASX7015E: Exception running command: "f.setMenuBar( m )";
8| exception information:
9| com.ibm.bsf.BSFException: exception from Jython: 10|Traceback (innermost last):
11| File "<input>", line 1, in ?
12|TypeError: setMenuBar(): 1st arg can't be coerced to 13| java.awt.MenuBar
14|wsadmin>
```

[2]The Swing JMenuBar methods are used exclusively in this book.

Let's take a quick look at how you can create an empty, yet colorful, menu bar so that you can more easily identify where it's placed. Figure 10-1 shows where the blue JMenuBar is positioned on the JFrame content pane. By selecting a colorful background, you don't have to add any items to the menu to see where it.

Figure 10-1. *Menu1 sample output with a blue JMenuBar*

Listing 10-4 shows how easily you can do this.

■**Note** You have to identify the preferred JMenuBar size (line 17), since it contains no JMenu entries. Otherwise no space would be allocated for the menu and it wouldn't be visible.[3]

Listing 10-4. The Menu1 Class Using Color to Show an Empty JMenuBar

```
7|class Menu1( java.lang.Runnable ) :
8| def run( self ) :
9| frame = JFrame(
10| 'Menu1',
11| size = ( 200, 125 ),
12| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 13| )
```

```
14| frame.setJMenuBar(
15| JMenuBar(
16| background = Color.blue,
17| preferredSize = ( 200, 25 )
18| )
19| )
20| frame.setVisible( 1 )
```

How will the application look with a couple of menu entries? Let's see.... Since the JMenuBar has a blue background color, let's make the menu entries use a color that contrasts well. Figure 10-2 shows a first attempt at doing this.

[3]I encourage you to comment out line 17 and test this statement yourself. Don't forget, however, to comment out the trailing comma on the previous line as well.

Figure 10-2. *Menu2 sample output with a JMenuBar and entries*

That's not quite what I expected. What happened to the white foreground color? Listing 10-5 shows the Menu2 class.

Listing 10-5. The Menu2 Class Specifying Foreground and Background Colors

```
9|class Menu2( java.lang.Runnable ) :
10| def run( self ) :
11| frame = JFrame(
12| 'Menu2',
13| size = ( 200, 125 ),
14| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
15| )
16| menuBar = JMenuBar(
17| background = Color.blue,
18| foreground = Color.white
19| )
20| fileMenu = JMenu( 'File' )
21| fileMenu.add( JMenuItem( 'Exit' ) )
```

```
22| menuBar.add( fileMenu )
23| helpMenu = JMenu( 'Help' )
24| helpMenu.add( JMenuItem( 'About' ) )
25| menuBar.add( helpMenu )
26| frame.setJMenuBar( menuBar )
27| frame.setVisible( 1 )
```

The reason the menu items don't show up well is that you specified the foreground color for the JMenuBar, *not* for the menu entries. If you specify the color of the menu items, you can see the expected contrast in colors. Figure 10-3 demonstrates this expected contrast.

Figure 10-3. *Menu3 sample output with colorful JMenuBar entries*

Listing 10-6 shows the Menu3 class being used to generate this output. It also shows you that building a hierarchy of menu entries can make your code a bit more challenging to read.

Listing 10-6. Menu3 Class with Contrasting Menu Entries

```
9|class Menu3( java.lang.Runnable ) :
10| def run( self ) :
11| frame = JFrame(
12| 'Menu3',
13| size = ( 200, 125 ),
14| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 15| )
16| menuBar = JMenuBar( background = Color.blue ) 17| fileMenu = JMenu(
'File', foreground = Color.white ) 18| fileMenu.add( JMenuItem( 'Exit' ) )
19| menuBar.add( fileMenu )
20| helpMenu = JMenu( 'Help', foreground = Color.white ) 21| helpMenu.add(
JMenuItem( 'About' ) )
22| menuBar.add( helpMenu )
23| frame.setJMenuBar( menuBar )
24| frame.setVisible( 1 )
```

Looking closely at Listing 10-6, you can find three different kinds of menu-related objects: JMenuBar: Created on line 16[4]•
• JMenu: Created on lines 17 and 20
• JMenuItem: Created on lines 18 and 21[5]

Looking again at Figure 10-3, you can see most of these distinct entries and you can use this to better understand these different menu classes. You can also see how the color scheme of the selected entries differs from the ones specified. The good news is that you don't have to describe this new scheme because it's already provided by the Java Swing classes. This is another example of the Swing library making your life as a GUI developer that much easier.

Unfortunately, not everything is this easy. Here is where you might get into a little bit of potential confusion based on names and terminology. If you ask non-programmers what the row of words across the top of an application is called, they are likely to call it a "menu," which makes sense.

For Swing developers, however, the row of words corresponds with the Java, or Jython, JMenuBar class instance and the words themselves are instances of the JMenu objects. In order to minimize confusion, this book consistently uses the Java Swing terminology. I'll call the group of words or terms the JMenuBar, whereas the individual objects are called JMenu entries.

As you can see, creating the JMenuBar and the JMenu entries, as well as the corresponding JMenuItems, requires a fair amount code. It's common practice to have a method that is responsible for the creating menu bars and their entries and items. This method makes your application constructor (or run() method) simpler, and therefore easier to read, understand, and maintain.

[4]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JMenuBar.html. [5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JMenuItem.html.

Reacting to Menu-Related Events

One of the nice things about menu items being a kind of abstract button is that you already know how to tell the "button" what to do when it is pressed. You can use the verbose Java addActionListen() method call or the actionPerformed keyword argument to identify the method to be called.

Consider this: It’s common practice for Java Swing developers to have the application class descend from the ActionListener class. This is frequently done using the syntax shown in Listing 10-7.

Listing 10-7. Java Class Implementing ActionListener

```
public myClass implements ActionListener { ...
JMenuItem menuItem = new JMenuItem( "Spam" );
menuItem.addActionListener( this );
...
publick void actionPerformed( ActionEvent e ) {

...
}
...

}
```

Using this technique can make the actionPerformed() method more complex than it really needs to be. If every button and menu item uses the same actionPerformed() method, this event handler needs to figure out which kind of event was used to initiate the call to the actionPerformed() method. Listing 10-8, on the other hand, shows how easy it can be, using Jython, to have a simple routine for each menu item action listener event handler.

Not only does this approach simplify each event handler, it also allows each method to have a name that is more appropriate for the event being handled.

Listing 10-8. Using the actionPerformed Keyword Argument

```
8|class Menu4( java.lang.Runnable ) :
9| def run( self ) :
10| frame = JFrame(

11| 'Menu4',
12| size = ( 200, 125 ),
13| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
14| )
15| menuBar = JMenuBar()
16| fileMenu = JMenu( 'File' )
17| exitItem = fileMenu.add(
18| JMenuItem(
```

```
19| 'Exit',
20| actionPerformed = self.exit
21| )
22| )
23| menuBar.add( fileMenu )
24| helpMenu = JMenu( 'Help' )
25| aboutItem = helpMenu.add(
26| JMenuItem(
27| 'About',
28| actionPerformed = self.about
29| )
30| )
31| menuBar.add( helpMenu ) 32| frame.setJMenuBar( menuBar ) 33|
frame.setVisible( 1 ) 34| def about( self, event ) : 35| print 'Menu4.about()' 36|
def exit( self, event ) : 37| print 'Menu4.exit()'
38| sys.exit()
```

Granted, in Java you don’t need the ActionListener actionPerformed() method to determine which event caused the routine to be called. You could make each JMenuItem use an anonymous ActionListener class implementation. Unfortunately, this approach is much more challenging to read, use, and maintain than the Jython technique used in the Menu4 class in Listing 10-8.

Using Radio Buttons on a Menu

Up to this point, the menu items have only included text. There are times, however, when it makes sense to use radio buttons on a menu to identify only selection. I didn’t have to look far to find an example of this. I often use the Notepad++ text editor, which has an Encoding menu item. The sub-menu on this entry includes five radio button menu items that identify the encoding. There is a special class for this in the Swing hierarchy, called JRadioButtonMenuItem.[6]

As you saw with radio buttons in Chapter 8, you need to be able to group all of the JRadioButtonMenuItem instances together so that the Swing framework can enforce one selected item limitation. Fortunately, you can use the same ButtonGroup class for the menu items that you used for radio buttons.

Figure 10-4. *Menu5 sample output with radio button menu items*

Figure 10-4 shows the sample output of the Menu5 class, which demonstrates the use of JRadioButtonMenuItem on the menu. In Listing 10-9, you can see most of the code needed to make this happen. The only routine that isn't shown is the one that creates and returns JMenuBar.

[6]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JRadioButtonMenuItem.htn

Listing 10-9. Menu5 Class that Uses Radio Button Menu Items

```
10|class Menu5( java.lang.Runnable ) :
11| def createMenuBar( self ) :
| ...
44| def run( self ) :
45| frame = JFrame(
46| 'Menu5',
47| size = ( 200, 125 ),
48| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
49| )
50| frame.setJMenuBar( self.createMenuBar() )
51| frame.setVisible( 1 )
52| def spam( self, event ) : print 'Menu5.spam()'
53| def eggs( self, event ) : print 'Menu5.eggs()'
54| def bacon( self, event ) : print 'Menu5.bacon()'
55| def about( self, event ) : print 'Menu5.about()'
56| def exit( self, event ) :
57| print 'Menu5.exit()'
58| sys.exit()
```

I think that having the menu-creation process in a method all to itself makes the

remainder of the code more understandable. Listing 10-10 shows the createMenuBar() method from this same class.

Listing 10-10 Menu5 createMenuBar() Method

```
11| def createMenuBar( self ) :
12| menuBar = JMenuBar()
13| fileMenu = JMenu( 'File' )
14| data = [
15| [ 'Spam' , self.spam ],
16| [ 'Eggs' , self.eggs ],
17| [ 'Bacon', self.bacon ]
18| ]
19| bGroup = ButtonGroup()
20| for name, handler in data :
21| rb = JRadioButtonMenuItem(
22| name,
23| actionPerformed = handler,
24| selected = ( name == 'Spam' )
25| )
26| bGroup.add( rb )
27| fileMenu.add( rb )
28| exitItem = fileMenu.add(
29| JMenuItem(
30| 'Exit',
31| actionPerformed = self.exit
32| )
33| )
34| menuBar.add( fileMenu )
35| helpMenu = JMenu( 'Help' )
36| aboutItem = helpMenu.add(
37| JMenuItem(
38| 'About',
39| actionPerformed = self.about 40| )
41| )
42| menuBar.add( helpMenu )
43| return menuBar
```

Notice that this code (see line 24) ensures that one of the radio buttons in the group is selected. It isn’t often that your event handlers (lines 52-58 in Listing

10-9) can actually be performed on a single line. But doing so shortens the listing somewhat.

Using Check Boxes on a Menu

As I'm sure you've already guessed, in addition to radio button menu items, you can also create check box menu items using the JCheckBoxMenuItem class.[7] Figure 10-5 shows some images of the same application using check box menu items.

Figure 10-5. *Menu6 sample output with check box menu items*

Now you can see another advantage of using a separate method to create the menu bar. Listing 10-11 shows only the createMenuBar() method from the Menu6 class because, other than this routine, Menu6.py and Menu5.py are the same.[8]

Listing 10-11. Menu6's createMenuBar() Class

```
10| def createMenuBar( self ) :
11| menuBar = JMenuBar()
12| fileMenu = JMenu( 'File' )
13| data = [
14| [ 'Spam' , self.spam ],
15| [ 'Eggs' , self.eggs ],
16| [ 'Bacon', self.bacon ]
17| ]
18| for name, handler in data :
19| fileMenu.add(
```

```
20| JCheckBoxMenuItem(
21| name,
22| actionPerformed = handler 23| )
24| )
25| exitItem = fileMenu.add(
26| JMenuItem(
27| 'Exit',
28| actionPerformed = self.exit 29| )
30| )
31| menuBar.add( fileMenu )
32| helpMenu = JMenu( 'Help' )
33| aboutItem = helpMenu.add(
34| JMenuItem(
35| 'About',
36| actionPerformed = self.about 37| )
38| )
39| menuBar.add( helpMenu )
40| return menuBar
```

[7]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JCheckBoxMenuItem.html.
[8]Except, of course, for the literal constants that contain the name of the script.

Separating Menu Items

Occasionally, it is convenient to use a horizontal line on a menu to separate groups of items. For example, Figure 10-6 shows how some check box menu items can be separated from one another. I think that you'll agree that the line makes them stand out significantly.

Figure 10-6. *Menu7 sample output showing menu item separation*

Interestingly enough, there are multiple ways that you can add such lines. Listing

10-12 shows three lines from the createMenuBar() method from the Menu7.py file. In it, you see these techniques and how they are used. In the first line, the familiar JMenu.add() method adds an anonymous JSeparator object.[9] This is such a simple and common operation that the JMenu class includes a method to do just this. In the second comment line, you see how the addSeparator() method adds a separator to the end of the specified JMenu. The third line shows that the JMenu has an insertSeparator() method that you can use to insert a separator line at a specified position of an existing menu.

[9]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JSeparator.html.

Listing 10-12. Menu7 JSeparator() and addSeparator()

viewMenu.add(JSeparator()) # Using JMenu.add() # viewMenu.addSeparator() # Using addSeparator() # viewMenu.insertSeparator(1) # Using insertSeparator()

Menu Mnemonics and Accelerators

If you look closely at the JMenuItem constructors, you may notice that one includes something called a keyboard *mnemonic*.[10] What is this all about? Well, it lets you specify a key that can be used to select the associated menu item when the menu is visible. To better understand this idea, you'll add a mnemonic to the same menu that you saw earlier. Figure 10-7 shows the Exit menu with an underscore under the letter "x." This indicates that the user can select this entry by pressing the "x" key.

Figure 10-7. *Menu8 menu items with a mnemonic on the Exit menu item*

Figure 10-7 shows what happens to the menu items for which mnemonic values are specified. In each instance, the first instance of the specified letter in the

menu item is underlined.[11] In this case, the “x” in Exit and the “A” in ANSI have menu shortcuts.

The mnemonic keystrokes are active only when the menu is being displayed. So, if you wanted to be able to use the “x” mnemonic to exit the application using the keyboard, you would need to press the F10 key to activate the first menu entry, and then press the “x” key to use the mnemonic to select this menu item.[12]

How do you define a mnemonic for a JRadioButtonMenuItem? Unlike the JMenuItem constructor, no parameter exists for a keyboard mnemonic. This is yet another example where Jython makes life so much easier for you. Instead of creating a JRadioButtonMenuItem and then using the setMnemonic() method inherited from the AbstractButton class, you can simply add a keyword argument to the constructor. Listing 10-13 shows how this is done in Menu8.py.

[10] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JMenuItem.html#JMenuIte

[11]In situations where you want to underline a different letter in the text string, use the setDisplayedMnemonicIndex() method inherited from the AbstractButton class, i.e., http://docs.oracle.com/javase/8/docs/api/javax/swing/AbstractButton.html#setDisplayedMnemonicIndex%28int%29.

[12]Of course, you could select the File menu entry using the mouse and then press the “x” key on the keyboard, but it’s just as easy to use the mouse to select Exit after using it to open the File menu.

Listing 10-13. Using the Mnemonic Keyword Argument on the JRadioButtonMenuItem Constructor

```
codeMenu = JMenu( 'Encoding' )
data = [
[ 'ANSI' , KeyEvent.VK_A ],
[ 'UTF-8' , KeyEvent.VK_U ],
[ 'UCS-2 Big Endian' , KeyEvent.VK_B ],
[ 'UCS-2 Little Endian' , KeyEvent.VK_L ]
]
bGroup = ButtonGroup()
for suffix, mnemonic in data :
name = 'Encoding in ' + suffix
```

```
35| rb = JRadioButtonMenuItem(
36| name,
37| mnemonic = mnemonic,
38| selected = ( suffix == 'ANSI' )
39| )
40| bGroup.add( rb )
41| codeMenu.add( rb )
```

One of the limitations of mnemonics is that the menu entry with a mnemonic has to be visible for the mnemonic key to be recognized. This brings us to the topic of to accelerator keys. The advantage of associating an accelerator key with a menu entry (or item) is that the menu does not have to be visible for the associated event to be initiated. One disadvantage of accelerator keys is that you need to use the menu item setAccelerator() method to associate an accelerator key with a menu item. Unless, of course, you’re using a wonderful language like Jython that allows you to use keyword arguments on your constructor calls.

Another possible disadvantage of accelerators is that, unlike mnemonics, there is no obvious indication that an accelerator key is in effect. Listing 10-14 shows a snippet from Menu9.py, which uses a bad practice of associating an unmodified “x” accelerator key with the Exit menu item. This means that when the user presses the “x” key, the program immediately exits. This is unlikely to be an event or action that will be anticipated, or appreciated, by your users.

Listing 10-14. Example of an Unmodified Accelerator

```
18| exitItem = fileMenu.add(
19| JMenuItem(
20| 'Exit',
21| KeyEvent.VK_X,
22| actionPerformed = self.exit,
23| accelerator = KeyStroke.getKeyStroke( 'x' )
24| )
25| )
```

What can and should be done about this? One possibility is to use the modified keys (Alt-X instead of simply X) .Menu10.py tries to do exactly this. Instead of “x” as shown in line 46 of Listing 10-14, Listing 10-15 defines the accelerator as Alt-X (see lines 23-27). Listing 10-15 is from Menu10a.py, which defines Alt-X

as the accelerator key for the Exit menu item.
Listing 10-15. Using Alt-X as an Accelerator

```
42| exitItem = fileMenu.add(
43| JMenuItem(
44| 'Exit',
45| KeyEvent.VK_X,
46| actionPerformed = self.exit,
47| accelerator = KeyStroke.getKeyStroke(
48| 'x',
49| InputEvent.ALT_DOWN_MASK
50| )
51| )
```

What does the menu item look like when you add this kind of accelerator key? Figure 10-8 shows how Swing displays this kind of information. I don't know about you, but I'm pretty impressed by Swing when I see that it automatically adds Alt-X to the menu item to clearly convey the presence of the accelerator key.

Figure 10-8. *Menu10 menu showing Alt-X accelerator key*

Unfortunately, this doesn't work for some reason. When you press Alt-X, nothing happens. Why not? Let's figure out what is wrong. What are the differences between the unmodified KeyStroke in Listing 10-14 (line 23) and the modified KeyStroke in Listing 10-15 (lines 47-50)? Listing 10-14 uses a KeyEvent constant instead a simple character string. Does it matter? Unfortunately, it does. Listing 10-16[13] shows what is returned by the KeyStroke.getKeyStroke() method using four different techniques.

Listing 10-16. Generating Modified KeyStroke Instances

```
wsadmin>from javax.swing import KeyStroke as KS
wsadmin>from java.awt.event import ActionEvent
wsadmin>from java.awt.event import InputEvent
wsadmin>from java.awt.event import KeyEvent
wsadmin>
wsadmin>KS.getKeyStroke( 'x' )
typed x
wsadmin>KS.getKeyStroke( KeyEvent.VK_X, ActionEvent.ALT_MASK )
alt pressed X
wsadmin>KS.getKeyStroke( KeyEvent.VK_X, InputEvent.ALT_MASK ) alt
pressed X
wsadmin>KS.getKeyStroke( 'x', ActionEvent.ALT_MASK )
alt typed x
wsadmin>
wsadmin>a = KS.getKeyStroke( KeyEvent.VK_X, ActionEvent.ALT_MASK )
wsadmin>i = KS.getKeyStroke( KeyEvent.VK_X, InputEvent.ALT_MASK )
wsadmin>a == i
1
wsadmin>
```

[13]Listing 10-16 uses the KS alias for KeyStroke so the lines aren't too long.

What does this mean for your applications? Well, it means that if you try to use the technique from Listing 10-15, line 48, your accelerator keys won't work. It isn't obvious from the user interface that there is a difference, but it's a subtle and important one at that. The last part of Listing 10-16 shows that you can use ActionEvent or InputEvent to identify the modification type—it doesn't matter. The result of the comparison is 1 (true), which means that the values are the same.

Pop-Up Menus

In addition to the menu bars, Swing applications can also use pop-up menus. What is a pop-up menu, exactly? Just as with a menu on the menu bar, when a pop-up menu is activated, its menu items are displayed. Unlike a menu bar menu, a pop-up menu is normally activated using a right-click on the component with which the pop-up menu is associated. Interestingly enough, this approach enables you to define multiple context-sensitive pop-up menus if that makes sense for your application.

Let's take a quick look at a trivial pop-up menu for a very simple application. Figure 10-9 shows the application output with two labels and two text input fields. If you right-click on either of the input fields, a small pop-up menu is displayed. Should you decide to select one of the values on the menu, that value is placed into the text field over which the right-click occurred.

Figure 10-9. *Popup1 sample output*

Listing 10-17 shows some of the Popup1 class methods used to produce the output shown in Figure 10-9.

Listing 10-17. Popup1 actionPerformed() and run() Methods

```
class Popup1( java.lang.Runnable ) :
 def actionPerformed( self, event ) :
 self.target.setText( event.getActionCommand() )
 def run( self ) :
 frame = JFrame(
 'Popup1',
 layout = GridLayout( 0, 2 ),
 defaultCloseOperation = JFrame.EXIT_ON_CLOSE
 )
 frame.add( JLabel( 'One' ) )
 frame.add(
 JTextField(
 5,
 mousePressed = self.PUcheck,
 mouseReleased = self.PUcheck
 )
 )
 self.PU = self.PUmenu()
 frame.add( JLabel( 'Two' ) )
 frame.add(
 JTextField(
```

```
32| 5,
33| mousePressed = self.PUcheck,
34| mouseReleased = self.PUcheck
35| )
36| )
37| frame.pack()
38| frame.setVisible( 1 )
```

Listing 10-18 shows the remainder of the Popup1 class methods found in the Popup1.py script.

Listing 10-18. Popup1 PUmenu() and PUcheck() Methods

```
39| def PUmenu( self ) :
40| def MenuItem( text ) :
41| return JMenuItem(
42| text,
43| actionPerformed = self.actionPerformed
44| )
45| popup = JPopupMenu()
46| popup.add( MenuItem( 'Spam' ) )
47| popup.add( MenuItem( 'Eggs' ) )
48| popup.add( MenuItem( 'Bacon' ) ) 49| return popup
50| def PUcheck( self, event ) :
51| if event.isPopupTrigger() : 52| self.target = event.getSource() 53|
self.PU.show(
54| event.getComponent(), 55| event.getX(),
56| event.getY()
57| )
```

One of the reasons for including multiple input fields was to demonstrate that the event used to display the popup can also be used to determine which component to associate with the event (see line 52 in Listing 10-18). In order for the pop-up menu to have access to this information, it needs to be saved somewhere in the application. Jython makes this really easy, by allowing you to dynamically create object attributes as they are needed. You can see that the only other occurrence of the self.target object attribute is found on line 13 of Listing 10-17 in the ActionListener actionPerformed() method. Table 10-1 describes the various parts of the Popup1 class.

Table 10-1. *Popup1 Class, Explained*

Lines Description

12-13 The actionPerformed() method is invoked when a pop-up menu item is selected (line 43).

14-38 Popup1 run() method, which:

· Creates the application frame (lines 15-19).

· Creates the labels and text fields (lines 20-36).

Note the use of the mousePressed and mouseReleased keyword assignments.

· Creates the pop-up menu (line 28).

· Resizes and shows the frame (lines 37-38).

39-49 PUmenu() method that's used to create the JPopupMenu and MenuItems.

50-57 PUcheck() method that's invoked by mouse-listener routines to verify that a pop-up trigger occurred. If a trigger did occur, it saves the associated input field and then shows the pop-up menu based on the cursor's location.

Summary

This chapter is all about menus and menu items—more specifically, it's about JMenus and JMenuItems. It is important to remember the difference. The former are AWT classes and the latter are Swing classes. You saw how easy it is to create menu bars, pop-up menus, and the event handlers that are associated with these menu items. In the next chapter, you take a look at really useful structure—JTree—which is great for displaying hierarchical relationships between information.

Chapter 11

Using JTree to Show the Forest: Hierarchical Relationships of Components

This chapter is all about building and using trees to display a hierarchical relationship between elements. Trees allow you to convey a specific relationship between different pieces of information. For example, this kind of structure is great for showing something like an organizational chart. By the time you're through with this chapter, you should not only be able to use trees in your own

applications, but you will also more aware of the capabilities and limitations of this useful structure.

Displaying the Servers in a WebSphere Application Server Cell

One of the really useful Swing structures is JTree,[1] which lets you show a hierarchical relationship of components. A simple WebSphere Application Server cell will have a number of nodes, with each node having a number of servers. This hierarchy is well suited for a tree structure. Figure 11-1 shows a sequence of images created by the Tree1.py script. Here's a description of each image, from top-left to bottom-right:

• This image only shows the cell name and the associated node names within the cell. The horizontal scroll bar shows that the tree is within a scroll bar, and that the available horizontal space isn't sufficient to display the widest node name.

• This image shows what the tree looks like with the first node expanded. The highlighting is an indication that a tree node is selected.
• This image shows what happens when the second node is expanded.
• The final image shows the application after it has been resized and has multiple nodes selected.

[1]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JTree.html.

Figure 11-1. *Tree1 sample output*

The resized window is large enough to display the complete tree, so no scroll bars are necessary. Trees display information vertically, with each line or row displaying a single piece of information, or data. Each of these data items is called a *node*, and the node at the top of the tree is called the *root node*.[2] A node may or may not have additional nodes beneath it. If it does, it is called a *branch node*, if it doesn't, it's called a *leaf node*. A branch node is also called the parent of the nodes beneath it, and they are referred to as children, or child nodes. You may have noticed that there is a bit of a terminology collision[3] here.

Figure 11-1 shows the name of the WebSphere cell as the root node, and the names of the WebSphere nodes as children of the root node of the tree. Subsequent images in this figure show how individual nodes can be selected, collapsed, or expanded using the mouse or keyboard.[4]

The second image in the figure shows an expanded branch node. In this case, the

first child node beneath the root identifies the WebSphere deployment manager node that has one child (leaf) node, identified as dmgr. In this application, each leaf node represents an application server and shows the name of each server. It's important to remember that server names must be unique within the same node, but can occur multiple times within the cell.
This hierarchical representation of the names of the WebSphere cell, nodes, and servers should match your conceptual understanding of the relationship between these WebSphere configuration objects.
As Figure 11-1 illustrates, it is very likely that the JTree instance will be in a JScrollPane to allow scroll bars to displayed as needed.

[2] To minimize the chance of confusion, I'm going to try to be consistent and refer to Swing tree nodes as nodes, and WebSphere Application Server nodes as WebSphere nodes.
[3]This topic also came up in Chapter 10.
[4]Tree nodes can be expanded or collapsed by positioning the entry using a mouse or cursor keys and pressing Enter.

Listing 11-1 shows the Tree1 class from the Tree1.py script used to generate the output shown in Figure 11-1.[5]
Listing 11-1. Tree1 Class Using the JTree Class

```
8|class Tree1( java.lang.Runnable ) :
9| def cellTree( self ) :
10| cell = AdminConfig.list( 'Cell' )
11| root = DefaultMutableTreeNode( self.getName( cell ) ) 12| for node in
AdminConfig.list( 'Node' ).splitlines() : 13| here = DefaultMutableTreeNode(
14| self.getName( node )
15| )
16| servers = AdminConfig.list( 'Server', node ) 17| for server in
servers.splitlines() :
18| leaf = DefaultMutableTreeNode(
19| self.getName( server )
20| )
21| here.add( leaf )
22| root.add( here )
23| return JTree( root )
24| def getName( self, configId ) :
25| return AdminConfig.showAttribute( configId, 'name' ) 26| def run( self ) :
```

```
27| frame = JFrame(
28| 'Tree1',
29| size = ( 200, 200 ),
30| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 31| )
32| frame.add( JScrollPane( self.cellTree() ) ) 33| frame.setVisible( 1 )
```

Table 11-1 provides a short description of the code used to build and display the hierarchical tree structure. From the code in Listing 11-1, the description in Table 11-1, and the output in Figure 11-1, it should be clear that the JTree class is easy to use and quite powerful. With no additional code, you can select one or more items as well as expand or collapse individual nodes.[6]

Table 11-1. *Tree1 Listing Explained*
Lines Description

9–23 cellTree() method that creates, populates, and returns a JTree instance representing the names of the WebSphere cell, nodes, and application servers. Each node in the tree is an instance of DefaultMutableTreeNode and may have zero or more child nodes (aka children).

24–25 getName() method that uses the AdminConfig.showAttribute() scripting object method call to obtain and return the name of the specified configuration object.
26–33 Tree1 run() method that creates the application frame, populates it, and displays it on the Swing event dispatch thread.

[5] Remember that AdminConfig is a wsadmin scripting object, and therefore is only available when wsadmin is used to execute the Jython script.
[6]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/DefaultMutableTreeNo

JTree Attributes and Methods

You can see, by looking at the cellTree() method in Listing 11-1, that it is really easy to create a JTree. You can also see, from Figure 11-1, that the default settings or attributes might not match your expectations. For example, you might not want to allow the users to select multiple entries. How do you limit the users to selecting zero or one item at a time? You'll see how to do this shortly. Before you do that, though, it is important to see how the tree structure is separated

from the data being represented.

The TreeSelectionModel Class

One scenario that occurs frequently in the Swing class hierarchy is having one class to display a structure, and having a related but separate class that holds the data. This is also the case for trees. The class used by a tree to hold its data is an implementation of the TreeModel interface.[7] The only implementation of this class used by this book is the DefaultTreeModel class.[8]

In addition to the data model, the JTree class also has a TreeSelectionModel associated with it. Unless, of course, you don't want to let the users select any of the tree nodes. If this is the case, it should be set to None (or null in Java terms).

What, exactly, does this mean? Does it mean that you can't expand or collapse the tree nodes? No, it doesn't. It just means that no node can be selected. Figure 11-2 shows images similar to those Figure 11-1, but this time none of the nodes can be selected.

Figure 11-2. *Tree2 sample output*

[7]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/TreeModel.html.

[8]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/DefaultTreeModel.htm

What do you need to do to set this kind of limitation? Well, it's really quite simple. Listing 11-2 contains the run() method from Tree2.py, which shows that you only need to change lines 32-34, which correspond to line 32 in Listing 11-1.

Listing 11-2. Tree2 run() Method

```
26| def run( self ) :
27| frame = JFrame(
28| 'Tree2',
29| size = ( 200, 200 ),
```

```
30| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
31| )
32| tree = self.cellTree()
33| tree.setSelectionModel( None )
34| frame.add( JScrollPane( tree ) )
35| frame.setVisible( 1 )
```

Let’s get back to the question raised earlier. How do you limit the selection to zero or one tree node? The answer is that you need to change the default selection mode using the setSelectionMode() method. What kinds of values does this method allow? The value must correspond to the TreeSelectionModel constants shown in Table 11-2.[9]

Table 11-2. *TreeSelectionModel Constants*

SINGLE_TREE_SELECTION Allows a maximum of one tree node to be selected.
CONTIGUOUS_TREE_SELECTION Allows a maximum of one continuous adjacent group of nodes to be selected.

DISCONTIGUOUS_TREE_SELECTION Allows an unlimited number of nodes to be selected, with no restriction that they be adjacent.

This table may make you think about lists and how you can limit the selection of list items.[10] In fact, there is a direct correlation between the selection settings allowed by JList instances and JTree instances. In the example scripts in Chapter 9, you used the JList selectionMode keyword argument to limit the JList instances to a single item.

You might wonder if you can do a similar thing with JTree instances. Listing 11-3 shows how you can use the classInfo function, which you learned about in Chapter 4, to display the JTree class hierarchy. It also shows all of the attributes that contain the string "selectionmode". Unlike the corresponding JList class hierarchy, the JTree class does not have a selectionMode attribute. So the answer to the question is no, you can’t do it quite that easily.

[9]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/TreeSelectionModel.htı
[10]See the section in Chapter 9 entitled “Limiting the Selectable Items.”

Listing 11-3 JTree class hierarchy showing selectmode attributes

```
wsadmin>from javax.swing import JTree
wsadmin>
wsadmin>classInfo( JTree, attr = 'selectionmode' ) javax.swing.JTree

selectionModel
| javax.swing.JComponent
| | java.awt.Container
| | | java.awt.Component
| | | | java.lang.Object
| | | | java.awt.image.ImageObserver
| | | | java.awt.MenuContainer
| | | | java.io.Serializable
| | java.io.Serializable
| javax.swing.Scrollable
| javax.accessibility.Accessible
wsadmin>
```

So, how can you do it? Well, Listing 11-4 contains the run() method from Tree3.py. The statement used to limit the tree selection to a single node is performed in lines 34-36, which correspond to line 33 in Listing 11-2. From this you can see that in order to change the selection mode you need to obtain the selection model used by this tree (via a call to the getSelectionModel() method as shown in line 34). You then call the setSelectionMode() method of this model to make the change.

Listing 11-4. Tree3 run() Method, Limiting Node Selection

```
27| def run( self ) :
28| frame = JFrame(
29| 'Tree3',
30| size = ( 200, 200 ),
31| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
32| )
33| tree = self.cellTree()
34| tree.getSelectionModel().setSelectionMode(
35| TreeSelectionModel.SINGLE_TREE_SELECTION
36| )
```

```
37| frame.add( JScrollPane( tree ) )
38| frame.setVisible( 1 )
```

■**Note** it is possible to use the Jython-provided attributes to minimize the explicit calls to the specified methods, but that is likely to cause confusion. Code to do this would look something like:

```
tree.selectionModel.selectionMode =
TreeSelectionModel.SINGLE_TREE_SELECTION
```

You will have to decide if using the attribute names like this instead of the getter and setter methods is worth the potential confusion that it might cause.

TreeSelectionListener

There are times when your applications will need to know when the user makes a tree node selection. For example, when the user selects a tree node it is likely that they want information about this node displayed immediately. You will need the application to display information about the selected WebSphere node as soon as the user selects the corresponding tree node.

To do this, you need to use the TreeSelectionListener interface.[11] Just like the ListSelectionListener,[12] which is described in Chapter 9, this interface has a single valueChanged() method. Can you use the valueChanged keyword argument on the JTree constructor call to identify a TreeSelectionListener method for the JTree instance? The simple quick answer to this question is yes. Figure 11-3 shows one possible use for this kind of listener.

Figure 11-3. *Tree4 sample output*

In Figure 11-3, you can see how the text field is updated by the tree selection listener event handler to reflect the data from the selected node; it also shows " <none>" if no node has been selected. This will prove to be very useful, as you'll see when you build on this application. Listings 11-5 and 11-6 show the Tree4 class that this application uses.

[11]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/TreeSelectionListener
[12]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/ListSelectionListener.

Listing 11-5. Tree4 Class Using a TreeSelectionListener

```
13|class Tree4( java.lang.Runnable ) :
14| def cellTree( self ) :
15| cell = AdminConfig.list( 'Cell' )
16| root = DefaultMutableTreeNode( self.getName( cell ) )
17| for node in AdminConfig.list( 'Node' ).splitlines() :
18| here = DefaultMutableTreeNode(
19| self.getName( node )
```

```
20| )
21| servers = AdminConfig.list( 'Server', node )
22| for server in servers.splitlines() :
23| leaf = DefaultMutableTreeNode(
24| self.getName( server )
25| )
26| here.add( leaf )
27| root.add( here )
28| return JTree( root, valueChanged = self.select )
29| def getName( self, configId ) :
30| return AdminConfig.showAttribute( configId, 'name' )
```

Listing 11-6 shows the rest of the Tree4 class, including the run() method and the TreeSelectionListener event handler. The select() method is the event handler method for this application. This routine is identified as the listener for the JTree instance by using the valueChanged keyword assignment on line 28 in Listing 11-5.

The run() method creates and populates the application frame. Lines 38-42 of Listing 11-6 show how easy it is to create the tree, limit the number of items that can be selected, and position it all on the application frame. This is accomplished in just three statements.

Setting up the rest of the frame is also as easy. Lines 43-52 show how four statements can create a label and a read-only text field. These are placed in a panel, which is then positioned on the bottom part of the frame.
This is a wonderful example of how easy it is to create an interesting and useful application using Swing components in a creative manner.

Listing 11-6. Tree4 Class Using a TreeSelectionListener, Continued

```
31| def run( self ) :
32| frame = JFrame(
33| 'Tree4',
34| size = ( 320, 200 ),
35| layout = BorderLayout(),
36| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
37| )
38| tree = self.cellTree()
```

```
39| tree.getSelectionModel().setSelectionMode(
40| TreeSelectionModel.SINGLE_TREE_SELECTION
41| )
42| frame.add( JScrollPane( tree ), BorderLayout.CENTER )
43| panel = JPanel()
44| panel.add( JLabel( 'Selected:' ) )
45| self.msg = panel.add(
46| JTextField(
47| '<none>', # Initial value 48| 20, # Field width (columns) 49| editable = 0 #
Disable editing 50| )
51| )
52| frame.add( panel, BorderLayout.SOUTH )
53| frame.setVisible( 1 )
54| def select( self, event ) :
55| tree = event.getSource()
56| if tree.getSelectionCount() :
57| node = str( tree.getLastSelectedPathComponent() ) 58| else :
59| node = '<none>'
60| self.msg.setText( node )
```

Table 11-3 describes the important changes from the previous tree examples and the Tree4 class shown in Listings 11-4 and 11-5. I think it's impressive how easily you can use the Swing classes to build this neat little application.[13]

Table 11-3. *Tree4 Listing Explained*

Lines	Description
28	The only real difference is where you specify the valueChanged keyword on the JTree constructor to specify the TreeSelectionListener event handler routine to be called.
35	To make the presentation more aesthetically appealing, the frame uses the Border Layout Manager.
42	You position the JScrollPane instance containing the JTree in the center of the application frame.
43–52	You create a Panel container to hold the label and the disabled text field on the bottom of the application frame.

54–60 The select() method is the TreeSelectionListener valueChanged event handler. This method uses the event.getSource() method to obtain a reference to the associated tree, and if a node has been selected, it uses the getLastSelectedPathComponent() method call to identify that node.

JTree Manipulation

Some applications provide a way for the users to make changes to the tree being displayed. For example, you might want your application to provide a way for the user to change the tree node to reflect new information. This can be as simple as allowing the displayed values to be modified, or it may be appropriate to add and remove specific nodes.

[13]The dataType of the value returned by the getLastSelectedPathComponent() method call is an object, so it needs to be converted to a string before this value can be used to update the JTextField value.

.Let’s break this example into pieces, and not just because the listing, even without comments and blank lines, is too big. To start, let’s see what the application looks like when the DynamicTree.py script is executed. Figure 11-4 shows three images of this application before any changes are made.[14]

Figure 11-4. *DynamicTree sample output*

The first part of the DynamicTree class, shown in Listing 11-7, provides enough information to understand the layout of the application shown in Figure 11-4. The top part of the application window has a scroll pane containing the initial tree. Beneath this, there is a row of buttons. The run() method, shown in lines 59-77, creates and populates the frame, as you have seen before. For now, don't worry about the constructor (the __init__() method) or the getSuffix() method. You'll see how they are used shortly.

[14]This example is based on the DynamicTreeDemo from the Java Swing Tutorial, which can be found at http://docs.oracle.com/javase/tutorial/uiswing/components/tree.html#dynamic.

Listing 11-7. DynamicTree Class (Part 1 of 5)

```
53|class DynamicTree( java.lang.Runnable ) :
54| def __init__( self ) :
55| self.nodeSuffix = 0
56| def getSuffix( self ) :
```

```
57| self.nodeSuffix += 1
58| return self.nodeSuffix
59| def run( self ) :
60| frame = JFrame(
61| 'DynamicTree',
62| layout = BorderLayout(),
63| locationRelativeTo = None,
64| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
65| )
66| self.tree = self.makeTree() # Keep references handy
67| self.model = self.tree.getModel()
68| frame.add(
69| JScrollPane(
70| self.tree,
71| preferredSize = Dimension( 300, 150 )
72| ),
73| BorderLayout.CENTER
74| )
75| frame.add( self.buttonRow(), BorderLayout.SOUTH )
76| frame.pack()
77| frame.setVisible( 1 )
```

The only new thing in this run() method is line 63, which you might not have seen. What does this keyword argument do? It corresponds with the setLocationRelativeTo() method from the JFrame class that is inherited from the java.awt.Window class.[15] By initializing this value to None, which corresponds to the Java null, the application window is positioned in the center of the screen.[16] Pretty neat, eh?

The buttonRow() method, called in line 75 of Listing 11-7, can be seen on lines 78-94 in Listing 11-8. There really shouldn’t be anything too surprising in this routine, which uses a GridLayout instance to position the buttons being created in one horizontal row and assigns the actionPerformed event handler for each button using a keyword argument (line 90).

Listing 11-8. DynamicTree Class (Part 2 of 5, buttonRow() Method)

```
78| def buttonRow( self ) :
79| buttonPanel = JPanel( GridLayout( 0, 3 ) )
```

```
80| data = [
81| [ 'Add' , self.addEvent ],
82| [ 'Remove', self.delEvent ],
83| [ 'Clear' , self.clsEvent ]
84| ]
85| self.buttons = {}
86| for name, handler in data :
87| self.buttons[ name ] = buttonPanel.add ( 88| JButton(
89| name,
90| actionPerformed = handler, 91| enabled = name != 'Remove' 92| )
93| )
94| return buttonPanel
```

[15]See http://docs.oracle.com/javase/8/docs/api/java/awt/Window.html#setLocationRela

[16]Actually, the top-left corner of the application is positioned on the center of the screen. This should be “close enough” for now.

Listing 11-9 shows the code for the three button event handlers that are referenced by the data array in lines 80-84 of Listing 11-8. For now, I’ll just say that an event handler method is called when the button is pressed. I’ll defer a discussion of these specific event handlers until a little later. For now, it makes sense to provide them here because of their references in the previous listing.

Listing 11-9. DynamicTree Class (Part 3 of 5, Button Event Handlers)

```
95| def addEvent( self, event ) :
96| sPath = self.tree.getSelectionModel().getSelectionPath()
97| if sPath : # Use selected node
98| parent = sPath.getLastPathComponent()
99| else : # Nothing selected, use root 100| parent = self.model.getRoot()
101| kids = parent.getChildCount()
102| child = DefaultMutableTreeNode(
103| 'New node %d' % self.getSuffix()
104| )
105| self.model.insertNodeInto( child, parent, kids ) 106|
self.tree.scrollPathToVisible(
107| TreePath( child.getPath() )
108| )
```

```
109| def delEvent( self, event ) :
110| currentSelection = self.tree.getSelectionPath() 111| if currentSelection :
112| currentNode = currentSelection.getLastPathComponent() 113| if
currentNode.getParent() :
114| self.model.removeNodeFromParent( currentNode ) 115| return
116| def clsEvent( self, event ) :
117| self.model.getRoot().removeAllChildren()
118| self.model.reload()
```

The DefaultTreeModel Class

Listing 11-10 shows the makeTree() method called in line 66 of Listing 11-7. It uses instances of the DefaultMutableTreeNode class, which you saw previously. What’s new in this method is how the root node of the tree is passed as an argument to the DefaultTreeModel class constructor (line 126).

Listing 11-10. DynamicTree Class (Part 4 of 5, the makeTree() Method)

```
119| def makeTree( self ) :
120| root = DefaultMutableTreeNode( 'Root Node' )
121| for name in 'Parent 1,Parent 2'.split( ',' ) :
122| here = DefaultMutableTreeNode( name )
123| for child in 'Child 1,Child 2'.split( ',' ) :
124| here.add( DefaultMutableTreeNode( child ) )
125| root.add( here )
126| model = DefaultTreeModel(
127| root,
128| treeModelListener = myTreeModelListener()
129| )
130| tree = JTree(
131| model,
132| editable = 1,
133| showsRootHandles = 1,
134| valueChanged = self.select
135| )
136| tree.getSelectionModel().setSelectionMode(
137| TreeSelectionModel.SINGLE_TREE_SELECTION
138| )
139| return tree
```

Don’t forget that the JTree instance doesn’t actually contain the data being displayed. It only provides a view of the data, which like most non-trivial Swing components, is provided by a component data model. In the case of the JTree class, it’s almost always an instance of the DefaultTreeModel class.[17]

What does the tree model provide? Well, a lot of methods are provided by this class. Unfortunately, I don’t have the time or space to investigate and describe how and when they are used. At this point, I will only take a look at a few of them, specifically the ones related to adding and removing TreeModelListeners. In fact, that’s exactly what is done in line 128. The DefaultTreeModel instance is then passed to the JTree constructor in line 131. At the same time, other keyword arguments make the tree modifiable (line 132) and make the root node handle visible (line 133). Note that the valueChanged keyword assignment argument (line 134) is used to identify the TreeSelectionListener event handler for this tree instance.

Listing 11-11 shows the TreeSelectionListener event handler method for this DynamicTree class application. It is only a little different from the one you saw earlier in this chapter. This example determines if a removable node has been selected. Only non-root nodes can be selected, so you want the Remove button to be enabled only if the user has selected a non-root node. To do this, the code determines if a node has been selected, and if so, how far down the tree this selected node is. If no node has been selected, the count will be zero. The root node, by definition, has a depth of 1. This information is used in line 148 to determine if the Remove button should be enabled.

[17]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/DefaultTreeModel.htm

Listing 11-11. DynamicTree Class (Part 5 of 5, the select() Method)

```
140| def select( self, event ) :
141| tree = event.getSource() # Get access to tree
142| count = tree.getSelectionCount()
143| sPath = tree.getSelectionModel().getSelectionPath()
144| if sPath : # How deep is the pick?
145| depth = sPath.getPathCount()
146| else : # Nothing selected
147| depth = 0
148| self.buttons[ 'Remove' ].setEnabled( count and depth > 1 )
```

The TreeModelListener Interface

Now you finally get to the topic mentioned in Listing 11-10 on line 128—the TreeModelListener interface.[18] As you have seen, listeners are used when certain events occur. In this case, the methods in this listener are called when tree nodes are added, changed, or removed. In addition, another method is called when the tree structure changes. It is important to note the difference between this listener and the TreeSelectionListener discussed earlier.

By creating a class based on this listener interface, you can monitor changes that occur in the tree. The question, though, is how? Well, the TreeModelListener class is set up as a main class and isn't nested inside the application class. This demonstrates how you can use the TreeModelEvent[19] provided with each of the TreeModelListener methods in order to detect any changes that have occurred.

Listing 11-12 shows the myTreeModelListener class, which includes some common, shared methods that locate specific parts of the tree model. They identify the parent or the specific node of interest.

Listing 11-12. The myTreeModelListener Class

```
19|class myTreeModelListener( TreeModelListener ) :
20| def getNode( self, event ) :
21| try :
22| parent = self.getParent( event )
23| node = parent.getChildAt(
24| event.getChildIndices()[ 0 ]
25| )
26| except :
27| node = event.getSource().getRoot()
28| return node
29| def getParent( self, event ) :
30| try :
31| path = event.getTreePath().getPath()
32| parent = path[ 0 ] # Start with root node 33| for node in path[ 1: ] : # Get
parent of changed node 34| parent = parent.getChildAt(
35| parent.getIndex( node )
36| )
37| except :
38| parent = None
```

```
39| return parent
40| def treeNodesChanged( self, event ) :
41| node = self.getNode( event )
42| print ' treeNodesChanged():', node.getUserObject() 43| def
treeNodesInserted( self, event ) :
44| node = self.getNode( event )
45| print 'treeNodesInserted():', node.getUserObject() 46| def
treeNodesRemoved( self, event ) :
47| print ' treeNodesRemoved(): child %d under "%s"' % ( 48|
event.getChildIndices()[ 0 ],
49| self.getParent( event )
50| )
51| def treeStructureChanged( self, event ) :
52| print 'treeStructureChanged():'
```

[18]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/TreeModelListener.ht
[19]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/TreeModelEvent.htm

One question that people sometimes ask is, “How do I edit a node?” The sequence of images in Figure 11-5 demonstrates this process. You begin with the tree expanded and the root node selected. The users indicate that they want to edit the node by triple-clicking the node (three left clicks in a short time interval).[20] The node text is temporarily replaced with a text field containing the selected node text. This allows users to easily replace the text when they start typing. Should they want to cancel the edit operation, they can simply press Escape. To complete the edit, thereby replacing the original node value with the new one, users simply need to press Enter. This is the point at which the TreeModelListener treeNodesChanged() method is invoked.

[20]It’s interesting to note that the specified node is expanded or collapsed as part of the normal double-click process. That’s why the tree is collapsed as part of this edit process.

Figure 11-5. *Editing tree nodes*

How in the world would someone know this? Where is this documented? Well, if you think about it, there has to be some editor associated with the tree in order for you to modify node values. A DefaultTreeCellEditor[21] is one of the many things provided by the JTree class. This editor determines how the change is triggered, as well as the kind of edit input field to be displayed.

In addition to a default tree cell editor, the JTree constructor will also provide DefaultTreeCellRenderer[22] to determine how the tree nodes will be displayed. The fact that these support classes are part of the Swing hierarchy makes the lives of Swing application developers significantly easier.

Creating your own replacements for either of these classes is well beyond the scope of this book. However, the topic of cell renderers and editors is discussed in Chapter 12, which covers tables.

Summary

This chapter was all about creating, displaying, and manipulating trees. One of the most important points to remember about the JTree class is that it does not actually contain the data; it simply displays the hierarchical data in a form similar to an outline. The next chapter discusses the JTable class used to display data in a tabular format.

[21]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/DefaultTreeCellEditor.l
[22]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/DefaultTreeCellRender

Chapter 12

Motion to Take from the Table: Building Tables

One of the most useful structures in graphical applications is the table format, when information is displayed in an organized arrangement of rows and columns. In fact, you have seen lots of examples of tables throughout this book. As a side note, it is interesting how often words have multiple meanings and uses. This is especially true in the technology industry. It can also be the case with natural languages such as English. If you look for the definition of the

word "table," you'll find that it is often used as a noun and occasionally as a transitive verb (i.e., to place on or take off of an agenda). It is this second form that was the source for the title of this chapter,[1] just to be different. A table is such a common idea that most people would be hard pressed to remember when the concept was first described to them. This chapter is all about how to build, display, control, manage, and manipulate information in tables.

Tables Can Be Really Easy

We use tables of information all the time. In fact, it was while I was working with a simple wsadmin Jython script that listed port numbers and EndPoint names for my WebSphere Application Server environment that I started thinking about displaying the information in a table. The name of the application was ListPorts.py and Chapter 22 is all about the iterative development of a Swing application that does this.

Over time, I've written many different versions of that ListPorts.py script file. So

many that it's hard to keep track of them all. I wrote a script to locate the various instances of this script on my system and then created a table of the file date, size, and locations. Using this information, I produced a script that displays this information in table form, as shown in Figure 12-1.

[1]See *Robert's Rules of Order*, Article VI, Section 35.

Figure 12-1. *Table1.py output images*

There are multiple images in this figure to show that the table columns can be resized and moved around easily. To resize a column, you simply drag the vertical bar between two column headings one way or the other. To reorder the columns, you drag a column heading (such as "Date") to the desired position. The other columns are reordered accordingly. All of these capabilities are provided by the JTable class.

It's also interesting to note how the right-most column, which isn't wide enough to display the complete directory path, uses an ellipse (. . .) to indicate that the data has been truncated. Just as in Listing 12-1, which shows that lines 10-39 have been left out.[2] The last image shows that the column headings stay in place, even when the data scrolls down to display other rows in the table.

Listing 12-1. Excerpts from Table1.py

```
7|class Table1( java.lang.Runnable ) :
8| def __init__( self ) :
9| info = r'''
|...
40|04/23/2008| 506|C:\IBM\WebSphere\V7 Notes\listports.py
41|'''
42| self.data = [ # list comprehension
43| line.split( '|' ) # each row is an array
44| for line in info.splitlines() # each line is a row
45| if line # ignore blank lines
46| ]
```

[2] Of course, the Java Swing table does this automatically for you, whereas I had to do this manually in Listing 12-1.

```
47| def run( self ) :
48| frame = JFrame(
49| 'Table1',
50| size = ( 300, 200 ),
51| locationRelativeTo = None,
52| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
53| )
54| headings = 'Date,size,Location'.split( ',' )
55| frame.add(
56| JScrollPane(
57| JTable( self.data, headings )
58| )
59| )
60| frame.setVisible( 1 )
```

How much code is required to provide all of this functionality? Almost none, as you can see by looking at the Table1 class in Listing 12-1. Line 57 contains the JTable[3] constructor.[4] In fact, no additional code was required to provide all of the functionality just described. It is all provided by the Java Swing class library.

Defaults Can Be Harmful to your . . . Mental Health

There's a great deal that can be said for simplicity. In other words, defaults can sometimes drive you crazy. Just remember though—you get what you pay for. By that, I mean it is extremely unlikely that you will be able to have a really useful application that uses all of the Swing class defaults.

Let's take another look at the simple table created by the Table1.py script. Did you notice that all of the columns in the first image are the same width?

Unfortunately, the data found in each column isn't all the same width. But the default settings allocate each column the same amount of column space and each column is expected to contain the same type of data (i.e., a string).

You'll investigate column manipulation a little later in this chapter (in the section entitled "Column Manipulation"). For now, don't worry about it. First, I describe other aspects of tables.

Picky, Picky, Picky. . . Selecting Parts of a Table

What parts of the table can be selected? As you can see in Figure 12-2, multiple groups of rows can be selected. This figure also shows that cell values can be edited. When a cell is edited,[5] the cell highlighting is distinctly different and the cell has a more pronounced border. One of the challenges with the defaults is that every cell within the table is considered a string, so (almost) any kind of character data can be entered. For example, while editing the selected cell in Figure 12-2, you could easily enter any text you want. The caveat is that some characters will cause the edit mode to terminate (such as the Tab and Enter keys).

[3] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JTable.html.

[4]As you have seen before, it is a really good idea to place a potentially large component, such as a JTable instance, in a JScrollPane, as shown in lines 56-58.

[5]Either by double-clicking on the cell or positioning the focus on a cell using cursor control keys and pressing the spacebar.

Figure 12-2. *Table1 output: row selection and editing*

This provides you with a quick overview about some of the aspects related to Swing tables. The rest of the chapter discusses tailoring them to act in ways that make sense.

Row, Row, Row Your . . . Table? Working with Rows

If you play with the Table1.py application, you'll see that when you click on any cell in the table, the whole row is selected. In Figure 12-2, you can see that multiple groups of rows can, by default, be selected.[6] How can you change the row selection behavior? That's easy; the JTable class has a row selectionMode property that uses the same constants used when you were working with lists and trees. Table 12-1 describes the possible selection values.

Table 12-1. *Selection Mode Constants*

Selection Mode Constant	**Description**
SINGLE_SELECTION	Zero or one row can be selected.
SINGLE_INTERVAL_SELECTION	One contiguous group of rows can be selected.
MULTIPLE_INTERVAL_SELECTION	The default mode allows multiple groups of rows to be selected.

You can see, by looking at the JTable class Java documentation, that there is a setter but not a getter for the selectionMode property.[7] Don't be confused by the fact that there is a getter and setter for the selectionModel property; this is not what we're talking about. The fact that the class provides only a setter method means that you can use the setter (table.setSelectionMode()) method or you can specify the selectionMode keyword argument when the table is constructed, as shown in Listing 12-2, which is from Table2.py.

[6] Press and hold the Ctrl key as you use the mouse to select or deselect an individual row. If you hold the Shift key while making a selection, you can select multiple contiguous rows.

[7] I looked all over, but have yet to find any explanation for the lack of getter for this table property.

Listing 12-2. Specifying the selectionMode Using a Keyword Argument

```
6|from javax.swing import ListSelectionModel as LSM | ...
56| frame.add(
57| JScrollPane(
58| JTable(
59| self.data,
```

```
60| headings,
61| selectionMode = LSM.SINGLE_SELECTION 62| )
63| )
64| )
```

Does the presence of the selectionMode attribute in the JTable class mean that you can access its value (the operation normally performed by a getter method)? No, it doesn’t. If you try to do that, you’ll get an exception like the one shown in Listing 12-3.

Listing 12-3. Write-Only Attributes

```
wsadmin>from javax.swing import JTable
wsadmin>
wsadmin>data = [ [ '1.1', '1.2' ], [ '2.1', '2.2' ] ] wsadmin>head = [ 'Uno', 'Dos' ]
wsadmin>
wsadmin>table = JTable( data, head )
wsadmin>
wsadmin>table.selectionMode
WASX7015E: Exception running command: "table.selectionMode";

exception information:
com.ibm.bsf.BSFException: exception from Jython:

Traceback (innermost last):
File "<input>", line 1, in ?
AttributeError: write-only attr: selectionMode

wsadmin>
```

What does that mean for your applications? Basically, it means that if your application needs to display and modify this attribute, it needs to provide some way to maintain the property value as well as monitor any changes made to it.

Selecting Columns

The JTable class has columnSelectionAllowed and rowSelectionAllowed attributes. Interestingly enough, the former doesn’t appear on the Javadoc page of the JTable class, but it does exist, as you can see in Listing 12-4.

Listing 12-4. JTable selectionAllowed Attributes

```
wsadmin>from javax.swing import JTable
wsadmin>
wsadmin>classInfo( JTable, attr = 'SelectionAllowed' ) javax.swing.JTable

columnSelectionAllowed, rowSelectionAllowed | javax.swing.JComponent
| | java.awt.Container
| | | java.awt.Component
| | | | java.lang.Object
| | | | java.awt.image.ImageObserver
| | | | java.awt.MenuContainer
| | | | java.io.Serializable
| | java.io.Serializable
| javax.swing.event.TableModelListener
| | java.util.EventListener
| javax.swing.Scrollable
| javax.swing.event.TableColumnModelListener | | java.util.EventListener
| javax.swing.event.ListSelectionListener
| | java.util.EventListener
| javax.swing.event.CellEditorListener
| | java.util.EventListener
| javax.accessibility.Accessible
| javax.swing.event.RowSorterListener
| | java.util.EventListener
wsadmin>
```

Selecting Individual Cells

If you look at the JTable documentation, you should be able to find the cellSelectionEnabled attribute. Notice that it has been obsolete since version 1.3. How do you allow your table cells to be selected? Well, a cell can be selectable only when both rowSelectionAllowed and columnSelectionAllowed are true.

Interestingly enough, both a getter and a setter are provided for the cellSelectionEnabled attribute. The setter is used to assign the specified (Boolean) value to the rowSelectionAllowed and columnSelectionAllowed attributes, and the getter returns true only if both attributes are true. Table 12-2 shows the values returned by the rowSelectionAllowed, columnSelectionAllowed, and cellSelectionEnabled getter methods after the statement or method call listed in column 1 of the table has been executed.

Table 12-2. *Row, Column, and Cell Selection Values*
Statement Executed Row Col Cell JTable() 1 0 0 setColumnSelectionAllowed(1) 1 1 1 setRowSelectionAllowed(0) 0 1 0 setRowSelectionAllowed(1) 1 1 1 setCellSelectionEnabled(0) 0 0 0 setCellSelectionEnabled(1) 1 1 1

One question that might come to mind, especially if you have seen the TableSelectionDemo application[8] from the Java Swing Tutorial website, is "Aren't the Row, Column, and Cell selection attributes affected by the selection mode?" The answer is no, they aren't. The developer of that application has added this association, but this is not something that the JTable class does. So, the values in Table 12-2 are the same, regardless of the table selectionMode setting. However, this does point out how developers can choose to enhance classes to provide functionality for their applications.

I Am the Very Model of a Modern Major General: Table Models[9]

One of the many things that you haven't read about yet about JTable instances is the fact that the displayed information isn't stored in the JTable. The table actually provides a view to the data. The JTable class isn't the first complex Swing class that you've seen that does this. In Chapter 11, you learned how the JTree class used a data model to hold the information and the class used to display it.

What are the advantages to separating it like this? For one, it allows the view (the JTable class) to do things like determine the column order to be displayed and to rearrange it, if necessary. In case you are interested, you can disable the movement of columns. Listing 12-5 shows how you can use the JTable getTableHeader() method to access the JTableHeader[10] instance for this table. You then call the setReorderingAllowed() method with a value of 0 (false) to indicate that column reordering is not allowed.

Listing 12-5. Disabling Column Reordering
table.getTableHeader().setReorderingAllowed(0)
Types of Table Models

What kinds of table models exist in the Swing class hierarchy for your applications to use? There is an AbstractTableModel[11] class as well as a DefaultTableModel.[12] I'll let you guess which class is used as the default should

one not be provided for your JTable constructor call.

[8] See http://docs.oracle.com/javase/tutorial/uiswing/examples/components/TableSelecti components/TableSelectionDemo.java.
[9]With thanks and homage to Gilbert and Sullivan for their marvelous works.
[10]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/JTableHeader.html.
[11]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/AbstractTableModel.h
[12]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/DefaultTableModel.ht

Looking at the DefaultTableModel class documentation, you'll find more than two dozen methods, as well as another dozen inherited from the AbstractTableModel base class. What do these methods allow you to do with your table data? Well, from the user interface perspective, you can do things like add, move, and remove rows of data. One of the most important methods, however, is the isCellEditable() method, which is used by the JTable class to determine whether users are allowed to modify data in the specified cell. Listing 12-6 shows just how easily this can be used.

Listing 12-6. Creating a Read-Only TableModel Class

```
9|class roTM( DefaultTableModel ) :
10| def __init__( self, data, headings ) :
11| DefaultTableModel.__init__( self, data, headings ) 12| def isCellEditable(
self, row, col ) :
13| return 0
14|class Table3( java.lang.Runnable ) :

|...
62| frame.add(
63| JScrollPane(
64| JTable(
65| roTM( self.data, headings ),
66| selectionMode = LSM.SINGLE_SELECTION
67| )
68| )
```

69|)

The interesting point about the table model is how it is used, automatically, by the JTable class to determine what information is to be displayed and how. How does the JTable class determine how the data is displayed? It calls the tableModel getColumnClass(...) method, inherited from the base AbstractTableModel class, which you can and should override.

Wait a minute, what does this mean? Previously, you learned that the default data type for each cell is a string. Really, it's an *object* that's represented as a string by default. You can and should provide your own table model getColumnClass(...) instance to identify the appropriate data type for each column. Why a column? One simplifying implementation choice made by the Swing designers was to guarantee that all data in a JTable column would be of a single type. This shouldn't be too much of a restriction though since you can choose to identify the data type of the column as an object and figure out how to display and manipulate the actual data.

Listing 12-7 shows a relatively simple table model descendent class that only allows the values in column one (the second column)[13] to be modified (every other column is read-only). It also includes the getColumnClass(...) method, which identifies the appropriate data type for each column.

Listing 12-7. My Table Model Class

13|class myTM(DefaultTableModel) :
14| def __init__(self, data, headings) :
15| info = []
16| df = DateFormat.getDateInstance(DateFormat.SHORT) 17| for date, size, path in data :
18| info.append(
19| [
20| df.parse(date),
21| Integer(size.strip().replace(',', '')), 22| String(path)
23|]
24|)
25| DefaultTableModel.__init__(self, info, headings) 26| def getColumnClass(self, col) :
27| return [Date, Integer, String][col]

```
28| def isCellEditable( self, row, col ) :
29| return col == 1
```

[13]Remember that Jython and Java both use zero origin array indexing.

Figure 12-3, shows the resulting output of this application (the source for which is in Table4.py). Is the way the information is displayed in the table what you expected? Perhaps, perhaps not. Did you notice how the middle column is now right-aligned, whereas all of the others are left-aligned? Why do you think that is?

Figure 12-3. *Table4 output*

Cell Renderers

Another important aspect of the complex JTable class is the fact that each data type has a renderer instance that determines how the information should be presented to the user. Unless you provide one, an instance of the DefaultTableCellRenderer[14] class will be used. It is this renderer instance that displays the Integer values in the middle column using right justification. What

kind of cell renderers are provided by Swing? Table 12-3 provides this information.

Table 12-3. *Data Type-Specific Cell Renderers* **Data Type** Boolean Number
Double
Float
Date
Icon
Object

Renderer Description
Displayed as a check box.
Displayed as a right-justified label.
Like Number, but the value is provided by a NumberFormat instance, which is locale-specific. Same as Double.
Displayed as a left-justified label formatted by a DateFormat instance using the SHORT variation. Displayed as a centered label.
Displayed as a left-justified string in a label field.

Figure 12-4 shows some application images using various data types. For this application, the default isCellEditable(...) method, that is, the one provided by DefaultTableModel, returns true for every cell. The first image shows the initial column widths, the next image shows how a simple date value can be entered, and the last two images show how a value of 1234567890 is displayed by the integer and float renderers. Notice how commas are present in the Float column, but not in the Integer column.

[14]See
http://docs.oracle.com/javase/8/docs/api/javax/swing/table/DefaultTableCellRend

Figure 12-4. *Table5 output: data type rendering*

Custom Cell Renderers

I have a question for you to consider: what do you think about representing a Boolean value as a check box? This might make sense, at least as long as the data is editable. It is also very likely that Boolean values will be editable.

While working on this chapter, I wondered if the selection model had any impact on the row, column, and cell selection values. So I wrote a little Swing application to answer this question. The results of this test are displayed in Table 12-2.

Three versions of the application can be found in the TableSelection#.py script files, where # is 1, 2, or 3. The first version uses the simple expedient of not providing a TableModel, so all of the table cells default to being an object, the default representation of which is a left-justified string value.

A reasonable improvement or iteration is to provide a simple table model that identifies each cell as being of type Boolean, which results in the values being displayed using a check box centered in the cell. That's exactly what the other versions of the application do. This is the point at which I wondered what it would take to provide my own renderer, one that displays the Boolean value as a 0 or a 1. That's what the third version of the application does. Since a simple table model class already exists to identify each cell as a Boolean, and since a default renderer for type Boolean already exists (see Table 12-3), you have to figure out how to replace this default Boolean renderer with one of your own, specifically one that displays each value as a digit, centered in the cell. Figure 12-5 shows images from all three verisons of this application.

Figure 12-5. *TableSelection output with default and custom renderer*

The first application, TableSelection1.py, simply creates each JTable instance using the defaults. The second image, generated by TableSelection2.py, uses a table model that identifies each cell as being of type Boolean. The last image, generated by TableSelection3.py, replaces the default renderer of Boolean values with the one shown in Listing 12-8 (line 81).

Listing 12-8: Custom Renderer for Boolean Values

```
| 17|class boolRenderer( DefaultTableCellRenderer ) :                    |
| 18|    def __init__( self ) :                                           |
| 19|        self.result = JLabel(                                        |
| 20|            horizontalAlignment = SwingConstants.CENTER              |
| 21|        )                                                            |
| 22|    def getTableCellRendererComponent(                               |
| 23|        self,                                                        |
| 24|        table,               # JTable  - table containing value     |
| 25|        value,               # Object  - value being rendered       |
| 26|        isSelected,          # boolean - Is value selected?         |
| 27|        hasFocus,            # boolean - Does this cell have focus?|
| 28|        rowIndex,            # int     - Row # (0..N)               |
| 29|        vColIndex            # int     - Col # (0..N)               |
| 30|    ) :                                                              |
| 31|        self.result.setText( value.toString() )                      |
| 32|        return self.result                                           |
|...| ...                                                                 |
| 81|        bTable.setDefaultRenderer( Boolean, boolRenderer() )        |
|...| ...                                                                 |
```

A Few Cautions. . .

As you can see in Listing 12-8 as well as in the Javadoc for the DefaultTableCellRenderer class and the TableCellRenderer interface[15] on which it is based, only one method class exists. When providing a custom TableCellRenderer, it is important that you realize the potential performance impacts that can occur because of how the getTableCellRendererComponent(...) method is used. So, take a few moments to read the “Implementation Note” on the DefaultTableCellRenderer documentation page.

One of the important things that a custom TableCellRenderer should do is reuse a component instance, instead of instantiating a new one on each call to the getTableCellRendererComponent(...) method. Listing 12-8 shows how this class creates a common JLabel component instance in the constructor and reuses it on each use (lines 31 and 32).

It’s interesting to note that if, for some reason, you don’t like the column-centric technique for choosing the cell renderer, it is possible to create a table class based on JTable, which provides a getCellRenderer(...) method that uses a different technique. However, as you might imagine, the difficulties associated with a drastic approach such as this is not something to be underestimated. Although possible, it should rarely be considered a viable option.

[15]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/TableCellRenderer.htm

Which Cell Renderer to Use?

For tables that contain a variety of data types, it is quite possible that multiple cell renderer instances exist. When this occurs, it might not be clear how the table determines which one to use. How is this determined? Well, if a renderer has been defined for the specified table column, it takes precedence. If not, the renderer for the data type for that column will be used. So, if you want to use a custom renderer, you have to decide if you want all cells of the same type to be rendered in the same fashion or if you only want the data in a specific column to use this custom renderer. This should help you decide which technique to use to specify the custom renderer.

What would be a good example of this kind of decision? Consider a situation whereby your table contains a number of columns containing floating-point values. You might want some of these rendered using one format (such as a percentage, with a specific number of decimal places) and another column as a completely different format (such as in currency).

You just saw how to set up a cell renderer for a specific data type. How do you set up a column-specific one? First, you need to access the TableColumn[16] instance for the specific table column in question. Fortunately, this is easily done by using the getColumn(...) method of the TableColumnModel[17] instance that is used to hold the column information for the specific table instance. This TableColumn class includes a setCellRenderer(...) method, which allows you to specify the renderer instance for this particular table column. Listing 12-9 shows how easy this is. Please note how the col variable identifies the column number for which this renderer instance is being specified.

Listing 12-9. Defining a Column-Specific Cell Renderer

```
| t = JTable( ... )                                                    |
| t.getColumnModel().getColumn( col ).setCellRenderer( myRenderer() )  |
```

Don’t JLabel Me

The custom renderer used in Listing 12-8 used an instance of the JLabel class. Can some other component class be used? Certainly! In fact, it appears to be quite reasonable to use something like the JFormattedTextField class that you first saw in Chapter 7.

Figure 12-6 shows some sample images of the first attempt to use a

JFormattedTextField as the component returned for the custom renderer of the last column of the table. This column, which contains values of type Double, has exactly two digits after the decimal point. Notice how the editor displays the cell data.

[16]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/TableColumn.html.
[17]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/TableColumnModel.h

Figure 12-6. *Table6a output with a custom renderer*

Unfortunately, it isn't quite right, is it? For example, when the first row is selected, the last cell isn't highlighted the same way that the rest of the row is. Additionally, the numeric values, although they are displayed with exactly two decimal places, are left aligned in the cell. Nor does the cell have a border when it is selected, as the other cells in the row do. Listing 12-10 shows the first attempt at a renderer, which uses a JFormattedTextField instance to display the values.

Listing 12-10. Table6a.py: First Attempt at Custom Renderer

```
class myRenderer( DefaultTableCellRenderer ) :
 def __init__( self ) :
 nf = NumberFormat.getInstance()
 nf.setMinimumFractionDigits( 2 )
```

```
22| nf.setMaximumFractionDigits( 2 )
23| self.result = JFormattedTextField( nf )
24| def getTableCellRendererComponent(
25| self,
26| table, # JTable - table containing value
27| value, # Object - value being rendered
28| isSelected, # boolean - Is value selected?
29| hasFocus, # boolean - Does this cell have focus?
30| rowIndex, # int - Row # (0..N)
31| vColIndex # int - Col # (0..N)
32| ) :
33| self.result.setValue( value )
34| return self.result
|...
74| model = myTM( self.data, headings )
75| table = JTable(
76| model,
77| selectionMode = ListSelectionModel.SINGLE_SELECTION 78| )
79| table.getColumnModel().getColumn(
80| model.getColumnCount() - 1 # i.e., last column 81| ).setCellRenderer(
82| myRenderer()
83| )
```

Let’s start by fixing the alignment problem. A little investigation into the JFormattedTextField Javadoc[18] shows that it inherits the horizontal alignment property from the JTextField class.[19] This allows you to add a horizontalAlignment keyword argument to the JFormattedTextField constructor call. Listing 12-11 shows this minor modification.

Listing 12-11. Table6b.py with Horizontal Alignment

```
19|class myRenderer( DefaultTableCellRenderer ) :
20| def __init__( self ) :
21| nf = NumberFormat.getInstance()
22| nf.setMinimumFractionDigits( 2 )
23| nf.setMaximumFractionDigits( 2 )
24| self.result = JFormattedTextField(
25| nf,
26| horizontalAlignment = JTextField.RIGHT
```

27|)

What does this do to the application's output? Figure 12-7 shows how this change affects the cells' appearance. This does improve things a little bit, but additional challenges remain.

[18]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFormattedTextField.html.

[19]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JTextField.html.

Figure 12-7.

FormattedTextField with horizontal alignment fix

You can use the fact that the DefaultTableCellRenderer already does a lot of the work for you to resolve another issue. With a little bit of code, you should be able to take advantage of this fact. Listing 12-12 shows one way to use the DefaultTableCellRenderer class to solve some of the custom rendering issues.
Listing 12-12. Table6c.py: One Solution to the Custom Rendering Issues

```
class myRenderer( DefaultTableCellRenderer ) :
 def __init__( self ) :
 nf = NumberFormat.getInstance()
 nf.setMinimumFractionDigits( 2 )
 nf.setMaximumFractionDigits( 2 )
 self.result = JFormattedTextField(
 nf,
 border = None,
 horizontalAlignment = JTextField.RIGHT
 )
 self.DTCR = DefaultTableCellRenderer()
 def getTableCellRendererComponent(
 self,
 table, # JTable - table containing value
 value, # Object - value being rendered
 isSelected, # boolean - Is value selected?
 hasFocus, # boolean - Does this cell have focus?
 row, # int - Row # (0..N)
 col # int - Col # (0..N)
 ) :
 comp = self.DTCR.getTableCellRendererComponent(
 table, value, isSelected, hasFocus, row, col
 )
 result = self.result
 result.setForeground( comp.getForeground() )
 result.setBackground( comp.getBackground() )
 result.setBorder( comp.getBorder() )
 result.setValue( value )
 return result
```

What does this mean as far as this output is concerned? Well, as you can see in

Figure 12-8 when a row is selected, all of the cells in the row have the same color scheme. The second image shows that when a cell in the last column is selected, a slightly darker border is visible. And finally, the last image shows that when an invalid value is specified, this fact is highlighted in a very obvious fashion. So it appears that using the default cell renderer is a viable solution for some of these display issues.

Figure 12-8. *Table6c output showing expected results*

Using Cell Editors

Another important part of displaying information in a tabular form is the fact that you'll often want to allow the user to modify the information. This is where a cell editor comes into play. For this purpose, the Swing hierarchy includes a CellEditor,[20] a TableCellEditor[21] interface, and a DefaultCellEditor[22] class. Which of these are important for providing your own cell editor? Well, the base CellEditor interface provides a group of methods that are rarely replaced or overridden. So, it is best if you just leave them alone.

On the other hand, the TableCellEditor interface provides a method called getTableCellEditorComponent(...) that enables developers to provide or specify an editor component with a table.

[20] See http://docs.oracle.com/javase/8/docs/api/javax/swing/CellEditor.html.
[21] See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/TableCellEditor.html.
[22] See http://docs.oracle.com/javase/8/docs/api/javax/swing/DefaultCellEditor.html.

Boolean Cell Editors

You've already seen some of the simple, default, cell editors. Consider for a moment what you are doing when you have a table display Boolean values with a check box. Every time you toggle the check box, you are in fact editing the value in that cell. That's why, if you have a table model class in your application and your table includes Boolean values, the setValueAt(...) method must save the value as a Boolean, and not as the Integer that is provided by the setValueAt(...) method. Listing 12-13 shows the simple table model class that was used to verify this fact. Notice how the setValueAt(...) method, in lines 27-29, displays the value and its data type before using the Boolean constructor to save the appropriate kind of value.

Listing 12-13. BoolEdit.py Table Model Class

```
13|class tm( DefaultTableModel ) :
14| def __init__( self ) :
15| head = 'Name,Value'.split( ',' )
16| self.data = [
```

```
17| [ 'False', Boolean( 0 ) ],
18| [ 'True' , Boolean( 1 ) ]
19| ]
20| DefaultTableModel.__init__( self, self.data, head )
21| def getColumnClass( self, col ) :
22| return [ String, Boolean ][ col ]
23| def isCellEditable( self, row, col ) :
24| return col == 1
25| def getValueAt( self, row, col ) :
26| return self.data[ row ][ col ]
27| def setValueAt( self, value, row, col ) :
28| print 'tm.setValueAt():', value, type( value )
29| self.data[ row ][ col ] = Boolean( value )
```

Figure 12-9 shows a couple of images of the application, as well as the output that is generated by the setValueAt(...) method. Note how the value that is provided to the method is a Python integer. This is why you need to convert it to a Boolean before saving it in the data array instance.

```
Press <Enter> to terminate the application:
tm.setValueAt(): 1 org.python.core.PyInteger
tm.setValueAt(): 0 org.python.core.PyInteger
tm.setValueAt(): 1 org.python.core.PyInteger
```

Figure 12-9. *BoolEdit sample output*

Did you notice that you didn't even have to provide a custom or specialized editor for this example? In Table 12-3, you can see how Swing provides a renderer for Boolean types. Since the possible values are very limited, you don't need to worry about an editor for Boolean types.

Numeric Cell Editors

What about the numeric types? What do the default renderers and editors for these types provide? Figure 12-10 shows some images from the sample NumbEdit.py application that show the minimum values for each type in the top rows and the maximum values for each type in the bottom rows.

Figure 12-10. *NumbEdit.py output: default numeric renderers and editors*

At first glance, it may seem that the minimum Double and Float values should be something other than zero. The fact that they are non-zero, just very small, as is evident when the default editor for either value is used. The second image shows that the minimum Double value is 4.9E-324, which is very close to 0. So

close, in fact, that the default renderer displays a value of zero instead. The third and fourth images show that the largest Double value has too many significant digits to be displayed in the available space (it has a value of 1.7976931348623157E308). so elipses are used to indicate that the values have been truncated.

The last image shows what happens when a user attempts to enter a value that is outside of the range of valid values. The default editor highlights the cell using a red border, and the user is unable to move the focus away from the cell.
One question that frequently comes to mind relates to the setValueAt(...) method that is called to save userspecified values of various types. Java examples frequently use a switch statement or nested if-then-else statements to deal with this type of thing. Jython, on the other hand, can make this kind of thing trivial, as shown in Listing 12-14.[23]

Listing 12-14. NumbEdit.py getValueAt(...) and setValueAt.(...) Methods

```
11|class tm( DefaultTableModel ) :
| ...
29| def getColumnClass( self, col ) :
30| return [ Byte, Double, Float, Integer, Long, Short ][ col ]
31| def getValueAt( self, row, col ) :
32| return self.data[ row ][ col ]
33| def setValueAt( self, value, row, col ) :
34| print 'tm.setValueAt():', value, type( value )
35| Type = self.getColumnClass( col )
36| self.data[ row ][ col ] = Type( value )
```

All of this capability is provided by the default numeric renderers and editors, which is actually pretty neat. Can you think of a situation where you might need to provide your own numeric editor? What about the situation where you want to allow a specific range of values? You might be able to use a numeric JSpinner as was discussed in Chapter 7, but let’s start with something a little simpler. What does it take to provide a custom numeric editor that verifies the user input before it is accepted?

Custom Numeric Cell Editors

First, you need to decide the range of value with which you want to work. For

example, say your application wants to work with a simple concept like TCP/IP port numbers, which range from 0 to 65535.[24] Since the maximum value is larger than Short.MAX_VALUE (i.e., 32767), you have to use an Integer type field, and the editor has to verify the user input before accepting it.

How, exactly, is the custom editor class supposed to check the user input? Well, first, you have to realize that the editor has to provide some sort of input field. If you base the custom editor on DefaultCellEditor,[22] you can use the fact that one of the constructors requires a JTextField. Listing 12-15 shows one way of doing this.

Listing 12-15. portEditor Class and Table Model from PortEdit.py

```
14|class portEditor( DefaultCellEditor ) :
15| def __init__( self ) :
16| self.textfield = JTextField(
17| horizontalAlignment = JTextField.RIGHT
18| )
19| DefaultCellEditor.__init__( self, self.textfield )
20| def stopCellEditing( self ) :
21| try :
22| val = Integer.valueOf( self.textfield.getText() ) 23| if not ( -1 < val < 65536 )
:
24| raise NumberFormatException()
25| result = DefaultCellEditor.stopCellEditing( self ) 26| except :
27| self.textfield.setBorder( LineBorder( Color.red ) ) 28| result = 0 # false
29| return result
30|class tm( DefaultTableModel ) :
31| def __init__( self ) :
32| head = 'Name,Value'.split( ',' )
33| self.data = [
34| [ 'Min Port', Integer( 0 ) ],
35| [ 'Max Port', Integer( 65535 ) ]
36| ]
37| DefaultTableModel.__init__( self, self.data, head ) 38| def isCellEditable(
self, row, col ) :
39| return col == 1
40| def getColumnClass( self, col ) :
41| return [ String, Integer ][ col ]
```

```
42| def getValueAt( self, row, col ) :
43| return self.data[ row ][ col ]
44| def setValueAt( self, value, row, col ) :
45| print 'tm.setValueAt():', value, type( value ) 46| self.data[ row ][ col ] =
Integer( value ) |...
55| table = JTable( tm() )
56| table.setDefaultEditor( Integer, portEditor() )
```

[23]Again, this script uses a temporary variable, Type, so the line isn't too long.
[24]Ignore the fact that a port number is in fact an *unsigned* short integer, which Java doesn't support.

This example shows how easy it is (line 27) to duplicate the technique used by DefaultCellEditors (that is, highlight the cell with a red border). It's also good to note that since the custom cell editor is based on a JTextField, the value provided to the setValueAt(...) table model method (lines 44-47) is a string, not an integer as you might expect.

How hard do you think it would be to change the code so it would display the port number using a different format (for example, in hexadecimal, octal, or even using a comma for values larger than 999)? Think about this for a moment. One significant decision you need to make is how, exactly, you want to represent port values. If you want to display (render) them using hexadecimal characters, should they be maintained, verified, and edited as character strings, or do you need to work with integer values elsewhere in the application?

These are the kinds of things that you need to consider as an application developer. How should the values be displayed (rendered) for the users? How do you expect the users to provided new values (editor)? What do you need to do to verify user input? Just think of the fun you have ahead of you.

JComboBox Cell Editors

If you look at the constructors for the DefaultCellEditor class,[22] you see one that accepts a JCheckBox argument (which is likely to be the one used by the default Boolean cell editor), one that accepts a JTextField argument, like the one you just saw, and one that accepts a JComboBox argument. What does that look like? Figure 12-11 shows some sample application images for the cbEdit1.py script that use a combo box editor for the values in column 1.

Figure 12-11. *cbEdit1.py output using a JComboBox editor*

It's interesting to note that you can't tell, from the initial display, that a combo box or drop-down list will be displayed, until you invoke the editor on the value displayed in column 1. Is there a way to show this? Certainly! All you have to do is set the cell renderer and the editor. Figure 12-12 shows some images from the cbEdit2.py application, which does this.

Figure 12-12. *cbEdit2.py output using a JComboBox editor and renderer*

A Slight Detour: Table Row Height

Figure 12-12 shows that the default might not always be optimal. So, how do you change the height of the table rows? When one of these kinds of questions arises, the first place that you should look is at the documentation for the class in question. In this case, the JTable[3] class. Search for "rowheight" and you'll find a (protected) rowHeight attribute, as well as some getter and setter methods.

The fact that the class includes multiple getter and setters, as shown in Table 12-4 illustrates that this class provides an opportunity to control the row height used by the rows in the table. In addition, each row can have a different height.

Table 12-4. *JTable rowHeight Getter and Setter Methods* **Getter/Setter Name**
getRowHeight()
getRowHeight(int row)
setRowHeight(int rowHeight)

setRowHeight(int row, int rowHeight)

Description
Returns the JTable rowHeight (in pixels).
Returns the rowHeight of the specified JTable row.

Specifies the rowHeight to be used by all table rows, in pixels, and initiates a table revalidation and repainting.
Specifies the rowHeight to be used by the specified row and initiates a table revalidation and repaint.

The results of using the rowHeight keyword attribute on the JTable constructor call is shown in Figure 12-13. Remember that since the frame containing the JTable instance is contained in a scroll pane, simply changing the height of the rows isn't sufficient. You also need to increase the height of the frame if you don't want the scroll bar to be displayed.

Figure

12-13. *cbEdit3.py output with slightly larger rows*

Interestingly enough, the second image shows that the height of the drop-down list (the ComboBox) isn't affected by the JTable row height. This shouldn't be too much of a surprise when you think about it. The row height that you changed was in the table, not in the ComboBox.

Listing 12-16 shows the slight modifications required to make the application easier to use. The only changes are found in lines 54, where the height of the frame is slightly larger (from 112 to 125 pixels). The other change can be seen in line 58, where you specify the rowHeight keyword argument used to assign the value of 20 (pixels) for each row. **Listing 12-16.** The cbEdit3 Class's run Method

```
51|class cbEdit3( java.lang.Runnable ) :
52| def run( self ) :
53| frame = JFrame(
54| 'cbEdit3',
55| size = ( 200, 125 ),
56| locationRelativeTo = None,
57| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
58| )
59| table = JTable( tm(), rowHeight = 20 )
60| table.setDefaultRenderer( JComboBox, cbRenderer() )
61| table.setDefaultEditor( JComboBox, cbEditor() )
62| frame.add( JScrollPane( table ) )
63| frame.setVisible( 1 )
```

Warning: Ugliness Ahead

Up to this point, I have been completely avoiding the fact that the version of Jython that is provided with the WebSphere Application Server product is, in a word, ancient. I'm sorry, but it's true. Up until now, it hasn't caused any problems, at least nothing significant. Now, however, you're going to see where it makes a difference.

When I tried to implement a simple cell editor based on the JSpinner[25] class, I converted a simple Java application, an abbreviated version of which is shown in Listing 12-17. The point that I want to make with this example is the fact that the

class extends the AbstractCellEditor class and implements the TableCellEditor interface, as you can see in lines 12 and 13.

Listing 12-17. SpinEditor.java JSpinner Cell Editor

```
12|public class SpinEditor extends AbstractCellEditor
13| implements TableCellEditor
14|{
15| JSpinner spinner;
16| public SpinEditor()
17| {
18| String values[] = { "Spam", "Eggs", "Bacon" };
19| spinner = new JSpinner( new SpinnerListModel( values ) );
20| spinner.setEditor( new JSpinner.ListEditor( spinner ) );
21| }
22| public Component getTableCellEditorComponent(
23| JTable table,
24| Object value,
25| boolean isSelected,
26| int row,
27| int column
28| )
29| {
30| spinner.setValue( value ); 31| return spinner;
32| }
33| public Object getCellEditorValue() 34| {
35| return spinner.getValue(); 36| }
37| public static void main(String[] args) 38| {
| ...
63| }
64|}
```

[25]SpinEditor.java is incompletely shown here; only enough is shown for discussion purposes.

Why use this approach? The main reason for doing so relates to the fact that the previous examples used the DefaultCellEditor constructors, as shown in Table 12-5. The Java example, shown in Listing 12-17, illustrates how you might go about creating a different kind of table cell editor, using a totally different

component type.

Table 12-5. *DefaultCellEditor Constructor Signatures*

DefaultCellEditor(JCheckBox checkBox) DefaultCellEditor(JComboBox comboBox) DefaultCellEditor(JTextField textField)

What does this look like when you convert it to Jython? You are likely to get an editor class similar to Listing 12-18. Notice how easy the conversion from Java to Jython is. This might be of assistance to you in the future, should you want to convert a Java Swing application to Jython.

Listing 12-18. SpinEdit1.py Editor Class

```
14|class editor( AbstractCellEditor, TableCellEditor ) :
15| def __init__( self ) :
16| values = 'Bacon,Eggs,Spam'.split( ',' )
17| self.spinner = JSpinner( SpinnerListModel( values ) )
18| self.spinner.setEditor(
19| JSpinner.ListEditor( self.spinner )
20| )
21| def getCellEditorValue( self ) :
22| return self.spinner.getValue()
23| def getTableCellEditorComponent(
24| self, # object reference
25| table, # JTable
26| value, # Object
27| isSelected, # boolean
28| row, # int
29| column # int
30| ) :
31| self.spinner.setValue( value );
32| return self.spinner;
```

Unfortunately, if you were to execute this script using the wsadmin utility from a WebSphere Application Server installation and click one of the values in the right-most column, you would see an exception like the one shown in Figure 12-14.

```
 Press <Enter> to terminate the application:Exception in thread
  "AWT-EventQueue-0" Traceback (innermost last):
  (no code object) at line 0
AttributeError: abstract method "isCellEditable" not implemented
```

Figure 12-14. *The wsadmin exception about the isCellEditable method*

The really strange thing is that when I tried to use this same script with the latest stable build of Jython (Jython 2.5.3), it worked just fine. But how do you get it to work with the version of Jython provided by the WebSphere product?

Instead of inheriting from AbstractCellEditor and TableCellEditor, as shown in Listing 12-17, you can base the editor class on the DefaultCellEditor class, as shown in Listing 12-19.

Listing 12-19. SpinEdit2.py Editor Class Based on DefaultCellEditor

```
13|class editor( DefaultCellEditor ) :
14| def __init__( self ) :
15| DefaultCellEditor.__init__( self, JTextField() )
16| values = 'Bacon,Eggs,Spam'.split( ',' )
17| self.spinner = JSpinner( SpinnerListModel( values ) )
18| self.spinner.setEditor(
19| JSpinner.ListEditor( self.spinner )
20| )
21| def getCellEditorValue( self ) :
22| return self.spinner.getValue()
23| def getTableCellEditorComponent(
24| self, # object reference
25| table, # JTable
26| value, # Object
27| isSelected, # boolean
28| row, # int
29| column # int
30| ) :
31| self.spinner.setValue( value );
32| return self.spinner;
```

This script works using the wsadmin utility and Jython 2.5.3. Since this book is primarily intended for WebSphere script writers, subsequent scripts will be based on the DefaultCellEditor class, and not on the AbstractCellEditor class and the TableCellEditor interface. Nonetheless, it was worth noting, so you don’t waste

the same kind of time that I did when I first encountered this issue.

Figure 12-15 shows some images from this application. The first two use an application frame height of 106 pixels, with each table row using 20 pixels. The next two images show what happens when each row uses 25 pixels and the frame height is set to 116 pixels.

Figure 12-15. *SpinEdit2.py output images*

The interesting note about these images is that when you use a uniform row height for each table row to better display the cell editor representation, the table representation might look quite right. This is due to the fact that you chose to display the cell values using a text, or string renderer, instead of a spinner renderer. If you use a custom renderer instead, you'll see that each cell in this column uses the spinner icons, as shown in Figure 12-16.

Figure 12-16. *SpinEdit3.py output images using a JSpinner renderer*

Did you notice how the cell being edited in the second image shows the text of the spinner selection in bold? It's little things like this that make your application more user friendly. That's one of the best reasons for using a robust and well-designed framework like Swing to build your applications.

Are there other improvements that you should make? Take another look at the application. In fact, let's make this more obvious by comparing the outputs of SpinEdit2 and SpinEdit3, side by side, when the first row in the table is selected. See Figure 12-17.

Figure 12-17. *SpinEdit2 and SpinEdit3 renderer differences*

Seeing them like this makes it more obvious that the custom renderer isn't dealing well with the cell colors. This shouldn't be too difficult to fix, right? Listing 12-12 showed how to solve a similar issue. Listing 12-20 shows the

revision of the sRenderer class from SpinEdit4.py.

Listing 12-20. SpinEdit4.py sRenderer Class

```
34|class sRenderer( DefaultTableCellRenderer ) :
35| def __init__( self ) :
36| self.DTCR = DefaultTableCellRenderer()
37| self.spinner = JSpinner( SpinnerListModel( choices ) )
38| def getTableCellRendererComponent(
39| self,
40| table, # table containing cell being rendered
41| value, # Object - value being rendered
42| isSelected, # boolean - Is value selected?
43| hasFocus, # boolean - Does this cell have focus?
44| row, # int - Row # (0..N)
45| col # int - Col # (0..N)
46| ) :
47| comp = self.DTCR.getTableCellRendererComponent(
48| table, value, isSelected, hasFocus, row, col
49| )
50| result = self.spinner
51| result.setForeground( comp.getForeground() )
52| result.setBackground( comp.getBackground() )
53| result.setValue( value )
54| return result
```

Unfortunately, this doesn't resolve the problem because the JSpinner class has its own model-specific editor. In order to "fix" this—that is, to have the spinner renderer reflect the appropriate colors—you need to change the text editor field that the spinner is using.

Listing 12-21 shows the revised spinner renderer class from the SpinEdit5.py sample application. Figure 12-18 shows what this does to the application's output.

Listing 12-21. SpinEdit5.py with Fixed sRenderer Class

```
34|class sRenderer( DefaultTableCellRenderer ) :
35| def __init__( self ) :
36| self.DTCR = DefaultTableCellRenderer()
```

```
37| self.spinner = JSpinner( SpinnerListModel( choices ) )
38| def getTableCellRendererComponent(
39| self,
40| table, # table containing cell being rendered 41| value, # Object - value being
rendered 42| isSelected, # boolean - Is value selected? 43| hasFocus, # boolean -
Does this cell have focus? 44| row, # int - Row # (0..N) 45| col # int - Col #
(0..N) 46| ) :
47| comp = self.DTCR.getTableCellRendererComponent( 48| table, value,
isSelected, hasFocus, row, col 49| )
50| tf = self.spinner.getEditor().getTextField()
51| tf.setForeground( comp.getForeground() )
52| tf.setBackground( comp.getBackground() )
53| self.spinner.setValue( value )
54| return self.spinner
```

Figure 12-18. *SpinEdit5 output*

Column Manipulation

Up to this point, you haven't learned much about column adjustments and manipulations. Back in Figure 12-1, you saw how the default table settings allow users to reorder the columns. Listing 12-3 shows how this feature can be disabled. Now, you're going to take a look at ways that you can manipulate your table columns.

Column Widths

In the section entitled, "Defaults Can Be Harmful to Your . . . Mental Health," you learned that, by default, the available horizontal space will be shared equally among each of the columns. If that is not the best appearance for your application, you need to take control of the way the column space is allocated. The first thing to realize is that your table might not have column headings. If it does, the information in these headings might not affect the way that column space is, or should be, allocated. For example, if the information in your column

headings is always wider than the table data, this can greatly simplify the way that column space should be allocated. On the other hand, if the data is sometimes wider than your column headings, you need to take this into account when your application determines how wide each column should be.

Column Heading Width

In order to determine the amount of space (the width) that should be used to comfortably display your headings, the program needs to make use of the renderer used by the table header. The fact that the table header has its own renderer shouldn't be too much of a surprise to you. Remember that the header will almost always contain strings, whereas the table data will have various types of data in each column. Besides, you previously learned about the JTableHeader class,[10] when you learned how to disable the reordering of columns.

Now, you're going to use this class to obtain the renderer used to display the table header. Let's revisit the application, last seen in Table6c.py, and revise it to adjust the width of the columns using the width of the column headings.

Listing 12-22 shows the code used to do this. You may be surprised by the row number of -1, in line 105. The Javadoc for the TableCellRenderer interface[15] includes a comment in the section on the getTableCellRendererComponent(...) method that states, "When drawing the header, the value of row is -1."

Listing 12-22. Table7.py: Specifying the Column Width Using Header Information Only

```
97| hRenderer = table.getTableHeader().getDefaultRenderer()
98| for col in range( model.getColumnCount() ) :
99| column = table.getColumnModel().getColumn( col ) 100| comp =
hRenderer.getTableCellRendererComponent( 101| None, # Table 102|
column.getHeaderValue(), # value 103| 0, # isSelected = false 104| 0, # hasFocus
= false 105| -1, # row # 106| col # col # 107| )
108| width = comp.getPreferredSize().width
109| column.setPreferredWidth( width )
```

This attempt fails because the preferred size is only a suggestion, not an absolute and firm limitation or restriction.[26] If you want to do this, you have to set the minimum and maximum column widths. Be careful though. In some classes the

method and attribute names have minimum and maximum spelled out entirely. The TableColumn class,[16] on the other hand, has setMinWidth(...), and setMaxWidth(...) methods instead. So, by replacing line 109 in Listing 12-20 with two statements, one to set the minimum column width and one to set the maximum column width, you get the output in Figure 12-19, which was generated using the Table8.py script.

[26]And thirdly, the code is more what you'd call "guidelines" than actual rules.

Figure 12-19. *Table8*

with fixed column widths
Determining Column Width

Well, that's a big failure. You certainly don't want to fix the column widths solely on the widths of the column headers, at least not for this kind of column headings that this application is using. Instead, you can process the data in the table, one column at a time, and compute the maximum preferred width for each value in the column. Then, you set the preferred width for the column using the maximum of the preferred header width and the maximum preferred width of all of the values in this column.

That doesn't sound too terribly bad. Are there any problems with this? Yes, unfortunately, there are. While working on a general function for this purpose, I encountered the following issues:
• The data returned by the getValueAt(...) method needs to be processed using the appropriate data type (the result of the getColumnClass(...) for the column). This wasn't too difficult to resolve since you can use the data type identified with this column to provide the renderer with the appropriate kind of value.

• Dealing with Date values can be somewhat of a challenge. In order to resolve this issue, you have to use the appropriate DateFormat instance that matches the way that your Date columns are displayed in the table. In this case, it seemed that the easiest way to deal with this was to use a DefaultRenderer and have it determine the preferred width of the string representation of the formatted date strings.

Listing 12-23 shows excerpts from the setColumnWidths(...) method of the Table9.py sample application. You might want to use something like this to determine how to allocate the column widths.
Listing 12-23. Excerpts from the setColumnWidths(...) Method from Table9.py

```
21|def setColumnWidths( table ) :
22| header = table.getTableHeader()
| ...
27| tcm = table.getColumnModel() # Table Column Model
28| data = table.getModel() # To access table data
29| margin = tcm.getColumnMargin() # gap between columns
30| rows = data.getRowCount() # Number of rows
31| cols = tcm.getColumnCount() # Number of cols
```

```
32| df = DateFormat.getDateInstance( DateFormat.MEDIUM )
33| tWidth = 0 # Table width
34| for i in range( cols ) : # For col 0..N
35| col = tcm.getColumn( i ) # TableColumn: col i
36| idx = col.getModelIndex() # model index: col i
37| render = col.getHeaderRenderer() # header renderer,
38| if not render : #
39| render = hRenderer # or a default
40| if render :
41| comp = render.getTableCellRendererComponent( | ...
48| )
49| cWidth = comp.getPreferredSize().width
50| else :
51| cWidth = -1
52| Type = data.getColumnClass( i ) # dataType: col i
53| for row in range( rows ) :
54| v = data.getValueAt( row, idx ) # value
55| if Type == Date :
56| val = df.format( v ) # formatted date
57| r = table.getDefaultRenderer( String ) 58| else :
59| val = Type( v )
60| r = table.getCellRenderer( row, i )
61| comp = r.getTableCellRendererComponent(
| ...
68| )
69| cWidth = max( cWidth, comp.getPreferredSize().width ) 70| if cWidth > 0 :
71| col.setPreferredWidth( cWidth + margin ) 72| tWidth +=
col.getPreferredWidth()
73| table.setPreferredScrollableViewportSize(
| ...
78| )
```

I think that you'll agree that the column widths shown in Figure 12-20 are a great improvement over the ones in Figure 12-19.

Figure 12-20. *Table9 with compute column widths*

Column Adjustments

If you have played with any of the table samples, you may have wondered why the columns react in the way that they do when you change the column widths. (Drag the vertical bar separating two column headers to the left or to the right to change the widths.)

Did you notice how all of the subsequent columns are adjusted to maintain the

table width? Why is this? Well, the JTable class includes a property called AutoResizeMode that the table uses to determine changes to the other columns' widths. Table 12-6 shows the constants used to modify this JTable property.

Table 12-6. JTable *Auto Resize Constants* **JTable Auto Resize Constant**
AUTO_RESIZE_OFF
AUTO_RESIZE_NEXT_COLUMN

AUTO_RESIZE_SUBSEQUENT_COLUMNS
AUTO_RESIZE_LAST_COLUMN AUTO_RESIZE_ALL_COLUMNS

Description
Columns will not be adjusted automatically.

Only the next column width will be affected by column width changes.

All subsequent columns will have their widths affected, in order to distribute the effects across any subsequent columns.
Note: This is the default setting.

Only the width of the last column will change.
All table columns will be adjusted to account for the userspecified width changes.

Let's see what this means in a real application. To start, disable the auto resize property of the table. Figure 12-21 shows the application when the AUTO_RESIZE_OFF and setColumnWidths(...) functions are used, as shown in Listing 12-21.

Figure 12-21. *Table10.py: computing column widths with auto resize off*

What would it take to have a simple table application that allowed you to dynamically change the auto resize mode, and then see how each setting affects the application? Figure 12-22 shows the results.

Table11
AUTO_RESIZE
T/F Date Integer Float Double
Aug 22, 2011 726 0 3.14
May 12, 2010 2535 1.1 6.28
Jun 23, 2009 1715 2.2 9.42
May 3, 2008 1697 3.3 12.57
Apr 23, 2008 506 4.4 15.71

Table11
AUTO_RESIZE
T/F Date Integer Float Double
Aug 22, 2011 726 0 3.14
May 12, 2010 2535 1.1 6.28
Jun 23, 2009 1715 2.2 9.42
May 3, 2008 1697 3.3 12.57
Apr 23, 2008 506 4.4 15.71

Table11
AUTO_RESIZE
T/F Date Integer Float Double
Aug 22, 2011 726 0 3.14
May 12, 2010 2535 1.1 6.28
Jun 23, 2009 1715 2.2 9.42
May 3, 2008 1697 3.3 12.57
Apr 23, 2008 506 4.4 15.71

Table11
AUTO_RESIZE
Off
Next
Rest
Last
All
Integer Float Double
726 0 3.14
2535 1.1 6.28
1715 2.2 9.42
1697 3.3 12.57
506 4.4 15.71

Figure 12-22. *Table11.py: dynamically selecting auto resize mode*

You can see that the application has a menu that allows you to select a different auto resize setting. Listing 12-24 shows parts of the Table11.py script that were used to produce the output in Figure 12-22. All of the radio button menu items identify the handler method as the action listener event handler. All this routine has to do is match the text of the user selection to the corresponding JTable auto resize constant and use this to change the auto resize mode on the table.

Listing 12-24. Selected Parts of Table11.py

```
130|class Table11( java.lang.Runnable ) :
| ...
144| self.info = [
145| [ 'Off' , JTable.AUTO_RESIZE_OFF ],
146| [ 'Next', JTable.AUTO_RESIZE_NEXT_COLUMN ],
147| [ 'Rest', JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS ],
148| [ 'Last', JTable.AUTO_RESIZE_LAST_COLUMN ],
149| [ 'All' , JTable.AUTO_RESIZE_ALL_COLUMNS ]
150| ]
| ...
185| def handler( self, event ) :
186| cmd = event.getActionCommand()
187| for name, value in self.info :
188| if cmd == name :
189| self.table.setAutoResizeMode( value )
190| self.table.repaint()
```

This kind of script is easy to produce, and it provides you with a simple test environment to better understand the implications of a table's property. In this case, you're looking at the auto resize property.

Summary

This chapter has illustrated some of the power and flexibility of the JTable class. It is important to remember that with great power there must also come great responsibility. This is just a reminder that you can quickly create a table in your application. However, if you do so, it is quite likely that the default settings are going to need some testing, manipulation, and tweaking in order to fit your exact

needs. As this chapter has shown, iterating your script with various enhancements isn’t that difficult. You are encouraged to use this technique to help improve your knowledge of tables and their characteristics.

Chapter 13

Keystrokes, Actions, and Bindings, Oh My!

There are a number of ways that graphical applications obtain user input. This chapter focuses on how applications deal with user input from the keyboard. This is all about what happens when the user presses a key, and how applications perceive and react to this kind of event.

Getting in a Bind: Looking at Bindings

In the section entitled “Menu Mnemonics and Accelerators" in Chapter 10, you learned how convenient it is to associate a particular keystroke[1] with an ActionListener[2] for menu items. This association is called a *binding*. Doing this allowed you to simplify the process of causing some menu-related action to occur. Thus, you were able to initiate the associated menu action using the specified keystroke.

In this chapter, you learn how other parts of your applications can be set up to react to specific keystrokes. You will also learn how to enhance the friendliness of your applications with these kinds of bindings.

What Is Meant by Binding?

Menu entries aren’t the only place in Swing where you can build an association between a keystroke and an action. In fact, creating this kind of association is called *binding*. A keystroke is bound to an action. One point you have to realize is that key bindings relate to a specific context. This has the potential of causing the users some confusion. For example, say you use a keystroke in a particular way for one context and in a different way in a different context. This might confuse or frustrate your users, so consider the consequences of such bindings carefully.

InputMaps and ActionMaps

How are bindings created? To begin, it is important to realize that the abstract JComponent class[3] includes actionMap and inputMap attributes, as you can see

in Listing 13-1.[4]

[1] Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/KeyStroke.html.
[2]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/ActionListener.html.
[3]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JComponent.html.
[4]The classInfo function can be found in code\Chap_04\classInfo.py.

Listing 13-1. JComponent Map Attributes

```
wsadmin>from javax.swing import JComponent wsadmin>
wsadmin>classInfo( JComponent, attr = 'map' ) javax.swing.JComponent

actionMap, inputMap
| java.awt.Container
| | java.awt.Component
| | | java.lang.Object
| | | java.awt.image.ImageObserver
| | | java.awt.MenuContainer
| | | java.io.Serializable
| java.io.Serializable
wsadmin>
```

Let's take a look at a specific class that descends from JComponent; for example, the JTable class.[5] If you use the classInfo function to display the methods that have the word "map" somewhere in their names, you get the output in Listing 13-2. This makes sense, since it shows that you have getter and setter methods for the inputMap and actionMap attributes. It is important to note, however, that the output of this function can be incomplete.

Listing 13-2. JTable Map Methods

```
wsadmin>from javax.swing import JTable
wsadmin>
wsadmin>classInfo( JTable, meth = 'map' )
javax.swing.JTable
| javax.swing.JComponent
> getActionMap, getInputMap, setActionMap, setInputMap | |
java.awt.Container
```

```
| | | java.awt.Component
| | | | java.lang.Object
| | | | java.awt.image.ImageObserver
| | | | java.awt.MenuContainer
| | | | java.io.Serializable
| | java.io.Serializable
| javax.swing.event.TableModelListener
| | java.util.EventListener
| javax.swing.Scrollable
| javax.swing.event.TableColumnModelListener
| | java.util.EventListener
| javax.swing.event.ListSelectionListener
| | java.util.EventListener
| javax.swing.event.CellEditorListener
| | java.util.EventListener
| javax.accessibility.Accessible
| javax.swing.event.RowSorterListener
| | java.util.EventListener
wsadmin>
```

If you look at the JTable Javadoc,[5] you'll find the inputMap and actionMap getters and setters listed in Table 13-1.

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JTable.html.

The interesting point is that there are two inputMap getters, one without an argument and the other with. What's that all about?

Table 13-1. *JTable inputMap and actionMap Getters and Setters* **Method return type and signature** InputMap getInputMap()

InputMap getInputMap(int condition)
void setInputMap(int condition, InputMap map) ActionMap getActionMap()

void setActionMap(ActionMap am)
Description
Returns the inputMap that is used when the component has focus.

Returns the inputMap that is used during condition. Sets the inputMap to use with the condition to map.

Returns the actionMap used to determine what action to fire for a particular KeyStroke binding.
Sets the actionMap to am.
The condition argument used by the second getter identifies which of three possible inputMaps should be returned to the caller. The constants that should be used are found in the JComponent class and are shown in Table 13-2.

Table 13-2. *JComponent Condition (Binding-Related) Constants* **Constant Name**
WHEN_FOCUSED
WHEN_ANCESTOR_OF_FOCUSED_COMPONENT

WHEN_IN_FOCUSED_WINDOW

Binding Context
Binding applies when the component has the focus.

Binding applies when the receiving component is an ancestor of the focused component or is itself the focused component.
Binding applies when the receiving component is in the window that has the focus or is itself the focused component.

So, it looks like you can use one of these JComponent constants as the condition argument to the getInputMap() method in order to retrieve the KeyStroke InputMap. But which constant is the right one to use?
You have a few options available. You could use all three and see which one results in inputMaps that contain information. Or you could take a look at the JComponent Javadoc[3] and read about the interesting and potentially useful methods that exist. Which ones, you might wonder? Well, the ones listed in Table 13-3 caught my attention (and aren't marked as obsolete).

Table 13-3. *JComponent Keystrokes and Actions Methods* **Method return type and signature Description** KeyStroke[] getRegisteredKeyStrokes() int getConditionForKeyStroke(KeyStroke aKeyStroke)
ActionListener getActionForKeyStroke(KeyStroke aKeyStroke)
Returns the keystrokes that will initiate registered actions. Returns the condition that determines whether a registered action occurs in response to the specified keystroke. Returns the object that will perform the action registered for a given keystroke.

Okay, I admit it. I didn’t know to look into the JComponent class for these methods until I used the classInfo function and it showed the KeyStroke methods. Listing 13-3 shows the result of doing this.

Listing 13-3. Keystroke Methods in JTable Hierarchy

```
wsadmin>from javax.swing import JTable
wsadmin>
wsadmin>classInfo( JTable, meth = 'keystroke' )
javax.swing.JTable

getSurrendersFocusOnKeystroke, setSurrendersFocusOnKeystroke |
javax.swing.JComponent
> getActionForKeyStroke, getConditionForKeyStroke >
getRegisteredKeyStrokes
| | java.awt.Container
| | | java.awt.Component
| | | | java.lang.Object
| | | | java.awt.image.ImageObserver
| | | | java.awt.MenuContainer
| | | | java.io.Serializable
| | java.io.Serializable
| javax.swing.event.TableModelListener
| | java.util.EventListener
| javax.swing.Scrollable
| javax.swing.event.TableColumnModelListener
| | java.util.EventListener
| javax.swing.event.ListSelectionListener
| | java.util.EventListener
| javax.swing.event.CellEditorListener
| | java.util.EventListener
| javax.accessibility.Accessible
| javax.swing.event.RowSorterListener
| | java.util.EventListener
wsadmin>
```

JTable Keystrokes

Let’s take advantage of these methods to see how many keystroke bindings exist for the JTable class. Listing 13-4 shows a simple script that can be used to

determine how many of these bindings exist, as well as determine the ones associated specifically with the spacebar.

Listing 13-4. The Keystrokes.py Script

```
1|from javax.swing import JTable
2|table = JTable()
3|keys = [
4| str( key )
5| for key in table.getRegisteredKeyStrokes()
6| ]
7|print 'Number of JTable KeyStrokes:', len( keys )
8|width = max( [ len( key ) for key in keys ] )
9|print 'JTable "Space" Keys:'
10|print '\n'.join(
11| [
12| '%*s' % ( width, key )
13| for key in keys if key.endswith( 'SPACE' ) 14| ]
15| )
```

The output of this script is shown in Figure 13-1. It contains a few interesting things to consider. First, notice that there are almost six dozen keystroke bindings for JTable instances. This should explain why the rest of the output is limited to instances of bindings for the spacebar key. The other point of interest is the way that the keystroke modifiers are presented.

```
Number of JTable KeyStrokes: 71 JTable "Space" Keys:
pressed SPACE ctrl pressed SPACE shift pressed SPACE shift ctrlpressed
SPACE
```

Figure 13-1. *Keystrokes.py output*

Additional tests of the keystroke modifiers show that each of the keystrokes has the four possible modifiers seen in Figure 13-1. This made me wonder how hard it might be to get the names of the actions for these keystrokes.
Listing 13-5 shows the KeyStrokes2.py script, which was written to answer these questions. **Listing 13-5.** KeyStrokes2.py Showing Bound Actions

```
1|from javax.swing import JTable
2|table = JTable()
```

```
keys = [
 key
 for key in table.getRegisteredKeyStrokes()
]
print 'Number of JTable KeyStrokes:', len( keys )
width = max( [ len( str( key ) ) for key in keys ] )
print 'JTable "Space" Keys:'
for key in keys :
 if str( key ).endswith( 'SPACE' ) :
 cond = table.getConditionForKeyStroke( key ) act = table.getInputMap( cond ).get( key ) print '%*s : %s' % ( width, str( key ), act )
```

The resulting output is shown in Figure 13-2. I don’t know about you, but I find this really fascinating.

```
Number of JTable KeyStrokes: 71
JTable "Space" Keys:
pressed SPACE : addToSelection ctrl pressed SPACE : toggleAndAnchor shift pressed SPACE : extendTo shift ctrl pressed SPACE : moveSelectionTo
```

Figure 13-2. *Keystrokes2.py output*

Putting It All Together

What would it take to create a simple application that displays a table of information showing the keystroke bindings for the JTable class? You can use a number of things that you’ve learned up to this point to do just that.

```
locationRelativeTo = None
```

While creating this application, you can also get a better understanding of something that you have been taking for granted up to now. Back in Chapter 11, you started seeing some examples that used the locationRelativeTo keyword argument to position the JFrame instance “in the middle of the screen.” So what exactly does that mean, and why am I bringing it up here?

Until this application, you didn't really have much of a reason to dig into it. This application changed that because it became more obvious what setting the locationRelativeTo attribute to None actually does. Essentially, using this setting causes the top-left corner of your application to be placed in the center of the screen. When your application isn’t too big for the screen, this is a very easy and convenient way to position the application window near the center of the screen.

Centering the Application

If you want to center the application's window in the middle of the screen, the application must be able to determine or compute the following:

- The size of the screen
- The size of the application (frame)

Then you have compute the best location for the top-left corner for the application using some simple math; so far, so good. The question is, what's the best way to determine the size of the screen? There's a simple answer to that question because that's something that GUI application developers have needed to know for a long time. Figure 13-3 shows just how easily this can be done.

Figure 13-3. *Determining screen size*

Unfortunately, you might not actually need to use this technique. Consider for a moment what using the locationRelativeTo keyword argument does. It places the application in the center of the screen. To correctly position the application, you only need to reposition it using the size of the application (which you can obtain using the JFrame getSize() method) and the center of the screen (which you can obtain using the JFrame getLocation() method). Listing 13-6 is an excerpt from the KeyBindings.py script; it shows one way that this can be done.

Listing 13-6. Centering an Application

```
134| size = frame.getSize()
135| loc = frame.getLocation()
136| frame.setLocation(
137| Point(
138| loc.x - ( size.width >> 1 ),
139| loc.y - ( size.height >> 1 )
140| )
141| )
```

The code expects that the JFrame instance is in the variable named frame, and that it has been properly populated and sized before the statement in line 134 is executed. It is important to realize what kinds of values are returned by the method calls in lines 134 and 135. The size value is an instance of the java.awt.Dimension class[6] (just like you saw in Figure 13-3). This makes perfect

sense since the application has a width and height. On the other hand, the call to the getLocation() method returns an instance of the java.awt.Point class,[7] because only the X and Y coordinates are needed to identify a point on the screen.

The call to the setLocation(...) method should be fairly clear, once you realize that the positioning adjustment requires that you move the X coordinate by half the width of the application and move the Y coordinate by half the height of the application. This is exactly what the expressions on lines 138 and 139 are doing.

[6]Seehttp://docs.oracle.com/javase/8/docs/api/java/awt/Dimension.html. [7]See http://docs.oracle.com/javase/8/docs/api/java/awt/Point.html. Defining the Table Properties

With a little thought, you'll likely realize that this table will be composed of strings and you need to prevent the users from being able to modify the table's contents. You can define a trivial table model containing only the getColumnClass(...) and isCellEditable(...) methods. Listing 13-7 shows just how simple this class can be.

Listing 13-7. Trivial Custom Table Model Class

```
75|class myTM( DefaultTableModel ) :
76| def getColumnClass( self, col ) :
77| return String
78| def isCellEditable( self, row, col ) :
79| return 0
```

Computing the Table Data

Most of the work needed to create this application has already been completed. The biggest challenge is using the stuff that you learned about earlier—using KeyStrokes.py and KeyStrokes2.py to build the two-dimensional array of strings. The first column should hold the keystroke name and then each of the columns can hold the action name for the corresponding keystroke-modifier column. Listing 13-8 shows the data(...) method, which builds and returns this table for the application.

Listing 13-8. KeyBindings.py Method to Build Table Data

```
80|class KeyBindings( java.lang.Runnable ) :
81| def data( self ) :
82| table = JTable() # use an empty (default) table
83| iMap = table.getInputMap(
84| JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT
85| )
86| keystrokes = [
87| ( key, iMap.get( key ) )
88| for key in table.getRegisteredKeyStrokes()
89| ]
90| keys = {} # Dict, index = key name -> modifiers
91| acts = {} # Dict, index = keyStroke -> actionName
92| for key, act in keystrokes :
93| val = str( key ) # e.g., shift ctrl pressed TAB
94| acts[ val ] = act # e.g., selectNextColumnCell
95| pos = val.rfind( ' ' )
96| prefix, name = val[ :pos ], val[ pos + 1: ]
97| if keys.has_key( name ) :
98| keys[ name ].append( prefix )
99| else :
100| keys[ name ] = [ prefix ]
101| names = keys.keys()
102| names.sort()
103| prefixes = [
104| 'pressed', # unmodified keystroke 105| 'ctrl pressed', # Ctrl-<keystroke>
106| 'shift pressed', # Shift-<keystroke> 107| 'shift ctrl pressed' # Shift-Ctrl-
<keystroke> 108| ]
109| result = [] # The 2D table of strings 110| for name in names : # For each key
name (e.g., TAB) 111| here = [ name ] # Current table row 112| for prefix in
prefixes :
113| kName = ' '.join( [ prefix, name ] ) 114| here.append( acts.get( kName, '' ) )
115| result.append( here )
116| return result
```

Table 13-4 explains how this method produces the desired result. It may help you to realize that the cells for which no bindings exist will contain empty strings.

Table 13-4. *KeyBindings.py data() Method, Explained*

Lines Description
82 Instantiates a JTable to simplify access to its methods.
83-85 Obtains an inputMap of the context used for all key bindings.

86-89 List-comprehension statement that builds a list of tuples for all the keystroke bindings and their associated action names.

90-100 Builds a dictionary, named keys, indexed by the key name (e.g., TAB) and identifying the modifier bindings. Also builds a dictionary, named acts, indexed by the keystroke (e.g., pressed TAB) containing the bound action.

101-102 Builds a sorted list of the bound (unmodified) keystroke names (e.g., SPACE and TAB).
103-108 Builds a list of the keystroke modifiers in column order.
109-116 Builds the table to be returned by the data() method (i.e., the two-dimensional array of strings).

The Fruits of Your Labor
Figure 13-4 shows the results of executing the KeyBindings.py application, before any row selection has been made.

KeyStroke	Unmodified	Ctrl	Shift	Shift-Ctrl
A		selectAll		
BACK_SLASH		clearSelection		
C		copy		
COPY	copy			
CUT	cut			
DELETE			cut	
DOWN	selectNextRow	selectNextRowChangeLead	selectNextRowExtendSelection	selectNextRowExtendSelection
END	selectLastColumn	selectLastRow	selectLastColumnExtendSelection	selectLastRowExtendSelection
ENTER	selectNextRowCell		selectPreviousRowCell	
ESCAPE	cancel			
F2	startEditing			
F8	focusHeader			
HOME	selectFirstColumn	selectFirstRow	selectFirstColumnExtendSelection	selectFirstRowExtendSelection
INSERT		copy	paste	
KP_DOWN	selectNextRow	selectNextRowChangeLead	selectNextRowExtendSelection	selectNextRowExtendSelection
KP_LEFT	selectPreviousColumn	selectPreviousColumnChangeLead	selectPreviousColumnExtendSelection	selectPreviousColumnExtendSelection
KP_RIGHT	selectNextColumn	selectNextColumnChangeLead	selectNextColumnExtendSelection	selectNextColumnExtendSelection
KP_UP	selectPreviousRow	selectPreviousRowChangeLead	selectPreviousRowExtendSelection	selectPreviousRowExtendSelection
LEFT	selectPreviousColumn	selectPreviousColumnChangeLead	selectPreviousColumnExtendSelection	selectPreviousColumnExtendSelection
PAGE_DOWN	scrollDownChangeSelection	scrollRightChangeSelection	scrollDownExtendSelection	scrollRightExtendSelection
PAGE_UP	scrollUpChangeSelection	scrollLeftChangeSelection	scrollUpExtendSelection	scrollLeftExtendSelection
PASTE	paste			
RIGHT	selectNextColumn	selectNextColumnChangeLead	selectNextColumnExtendSelection	selectNextColumnExtendSelection
SLASH		selectAll		
SPACE	addToSelection	toggleAndAnchor	extendTo	moveSelectionTo
TAB	selectNextColumnCell		selectPreviousColumnCell	
UP	selectPreviousRow	selectPreviousRowChangeLead	selectPreviousRowExtendSelection	selectPreviousRowExtendSelection
V		paste		
X		cut		

Figure 13-4. *KeyBindings.py's output, with no row selected*

One of the really neat things about this application is that you can use what you

see to understand the results of using these bindings. For example, take a look at the TAB row and the Unmodified column. The action that is seen here is selectNextColumnCell and the one in you can see that when you press either the Tab or the Shift-Tab key, the table selection moves in the expected fashion.

Take a look at the actions associated with the Ctrl-A and Ctrl-BACK_SLASH keys. Try pressing these keys and see if the results meet your expectations. You should realize that not all of the actions will be allowed. Why not? Well, some of them require that the current cell be editable (e.g., cut or paste). Others depend on the current values of the columnSelectionAllowed and rowSelectionAllowed table attributes. In spite of that, I find the output of this application quite useful. I hope that you do as well.

Binding Reuse

You might be wondering what you can do with this information. I'll get to that, but first I want to explain how I came to investigate keystroke bindings.

I was working with an application that used some JTable instances. One of the challenges that I encountered was that my tables had some read-only and read-write columns. I expected that the users of the application might want to skip from one read-write cell to another using the Tab and Shift-Tab keys. In order to do this, I first had to figure out how the current keystroke bindings worked.

Looking again at Figure 13-4, you can see that the Tab and Shift-Tab keys correspond to selectNextColumnCell and selectPreviousColumnCell, respectively. What can be done with this information? Well, what you want to do is have the Tab and Shift-Tab keys be bound to actions something like selectNextEditableColumnCell and selectPreviousEditableColumnCell.

I don’t know about you, but I would much prefer to use existing code instead of figuring out how to rewrite these two ActionListeners. That should make sense. How should you go about doing that? You have a number of options from which to choose. Let’s start with something simple and decide if and when you can improve things.

Where to Begin: Finding the Appropriate Action Class

With what kind of class do you need to start? Let’s see if you can get by with

something easy, like the AbstractAction class.[8] According to the Javadoc for that class, “The developer need only subclass this abstract class and define the actionPerformed method.” This sounds ideal for this situation.

The next question you need to ask is, “What will our class instance need to access in order to perform its role?” Well, since you are specifically concerned with JTable actions, it would seem likely that the action will need to know with which table instance the action should be working. Additionally, it will need to know the action it will be using to perform its role. In the case of selectNextEditableColumnCell, it needs to know that it will need to use selectNextColumnCell. Or will it? Do you really need to create a completely new action and replace the keystroke binding of the Tab key from selectNextColumnCell to a completely new action? What if your new action simply identifies itself as the original action and uses the original action to perform its role? That seems like a reasonable and fairly simple approach. It also simplifies the code that you need to write to perform the desired action.

What Do You Need to Worry About? Boundary Conditions

Whenever you create something like this, you need to be concerned with *boundary conditions.* What are those in this case? Well, consider a situation in which you have a read-only table. What should happen when the selectNextEditableColumnCell action is invoked? If you’re not careful, you might create an infinite loop situation, which would be a bad thing. You need to be certain that your action deals well with a read-only table.

What’s the next worst-case scenario? In my mind, it would be a table with only one editable cell. In that case, how many cells would your action need to check in order to find the next editable cell? Well, that would depend on the size of the table. Can you figure that out? Given access to the table, you can use the table.getRowCount() and table.getColumnCount() methods to determine how many cells exist.

Listing 13-9 shows a simple implementation of this approach. It includes a call to the default toolkit beep() method if no editable cells are found in the table. You can test this application, which can be found in the code\Chap_13\WoT.py script. You may want to modify the isCellEditable(...) method to return a 0 (false) in order to test the read-only table scenario.

[8]See http://docs.oracle.com/javase/8/docs/api/javax/swing/AbstractAction.html.

Listing 13-9. findEditableCell Class from WoT.py

```
39|class findEditableCell( AbstractAction ) :
40| def __init__( self, table, action ) :
41| self.table = table
42| self.original = table.getActionMap().get( action )
43| self.table.getActionMap().put( action, self )
44| self.beep = Toolkit.getDefaultToolkit().beep
45| def actionPerformed( self, actionEvent ) :
46| table = self.table
47| numCells = table.getRowCount() * table.getColumnCount()
48| for cell in range( numCells ) :
49| self.original.actionPerformed( actionEvent )
50| if table.isCellEditable(
51| table.getSelectedRow(),
52| table.getSelectedColumn()
53| ) :
54| return
55| self.beep()
|...
70| findEditableCell( table, 'selectNextColumnCell' )
71| findEditableCell( table, 'selectPreviousColumnCell' )
```

Summary

This chapter explained how Java Swing applications deal with keystrokes. Rather than have the application monitor the user input, the environment determines the action to be performed for the keystroke event based on the current context. It is important to know how this works so your applications can take advantage of the existing infrastructure.

The next chapter discusses events and event handlers.

Chapter 14

It’s the Event of the Year: Events in Swing Applications

I have been referencing events, of all sorts, in many of the preceding chapters. It seems like a good time to take a better look at how events are handled in Swing

applications.

This chapter is all about events, such as mouse clicks, and preparing an application to handle these kinds of events when they occur. You'll also see how easy it is in Jython to associate a method with an event. It is important to note that methods that are called when specific events occur are called *listeners*. The association of a listener method with a specific kind of event is a kind of registration. Let's begin this chapter by taking a look at the number of classes that have something to do with events and listeners.

If an Event Occurs and No One Hears It . . .

If you look at the "complete list of Java classes,[1] " you'll find over 4,000 items. Of these, almost 12 dozen have "event" somewhere in their name and another seven dozen have "listener" in their name. What are all these things?

You've seen a number of them already. For example, when you were working with the JButton class,[2] you learned that in order to be able to react to user input, you needed an ActionListener[3] or some descendant class instance containing an actionPerformed(...) method. Until you did this association, no method would be called when the event occurred and the events were lost.

What type of argument is supplied to the actionPerformed(...) method? An ActionEvent,[4] that's what. An ActionEvent is an object instance that is used to indicate that some kind of component action has occurred. It can be used by an event handler to do things like identify the component that generated the event. Since ActionEvents are passed to registered listening methods, this chapter focuses more on listeners than on the details about ActionEvents.

Many of the Java Swing examples that you'll investigate are almost certain to include a listener of some type, as well as the method it defines or requires. The interesting thing about many of the Jython script examples that you've seen is that they don't have to explicitly identify, include, and use the particular listener class that is needed for the kind of event to be handled. You simply use the appropriate keyword argument to identify the method to be called when the event occurs.

[1] See http://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html.

[2]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JButton.html.
[3]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/ActionListener.html.
[4]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/ActionEvent.html.

Is this always a good thing? Maybe, maybe not. Consider for a moment the perspective of an experienced Java Swing developer. Their experience may have trained them to look at Swing applications in a specific way to locate event handlers. Their first instinct may be to search for calls to the add*Listener(...) method. Are they going to locate them in Jython scripts that use the appropriate keyword argument? I don't think so, do you?

Personally, I find the use of the keyword argument to be a better approach for a number of reasons, including (but not limited to) these:

• It forces the developer to identify the event handler during component instantiation.

• It allows the event handler routine to be named something other than the generic actionPerformed(...) method name. For example, I prefer to see a method named buttonPressed(...) rather than the ubiquitous actionPerformed(...) variation, don't you?

• It allows multiple event handlers to be defined as part of the same application class. In this way, each button instance can have its own unique event handler, instead of having to share a single actionPerformed(...) method that must determine how it was invoked. This allows each event handler method to be simpler and uncluttered by code that needs to determine or identify the exact source of the associated event that caused the routine to be called.

• It allows the developer to avoid using multiple inheritance in their application class (for example, to include the ActionListener as one of its base classes).

Are there times when it might be a bad idea to use the keyword argument to identify the event handler to be invoked? Think about it for a moment. What does a method name like add*Listener(...) imply? It tells you that there are situations where multiple listeners may be appropriate. If this is the case then you are likely to be better served by using add*Listener(...) to identify all of the event handlers to be registered as event listeners. Otherwise, using the keyword argument approach may very well be the best. I'll talk more about this later in

this chapter.

Using Listener Methods

Most of the examples that you've seen have focused on a small number of listeners.[5] For example, in all of the JButton examples, you have only seen the ActionListener and its actionPerformed(...) method. Is that the only listener available for the JButton class? If you think so, you are sadly mistaken and in for a big surprise. Listing 14-1 shows the various listener methods and class hierarchy where each is defined.

Listing 14-1. JButton Listeners

```
wsadmin>from javax.swing import JButton
wsadmin>
wsadmin>classInfo( JButton, meth = 'listener' )
javax.swing.JButton
| javax.swing.AbstractButton
> addActionListener, addChangeListener, getActionListeners >
getChangeListeners, getItemListeners, removeActionListener >
removeChangeListener
| | javax.swing.JComponent
> > addAncestorListener, addVetoableChangeListener > > getAncestorListeners,
getVetoableChangeListeners > > removeAncestorListener,
removeVetoableChangeListener | | | java.awt.Container
> > > addContainerListener, addPropertyChangeListener > > >
getContainerListeners, removeContainerListener | | | | java.awt.Component
> > > > addComponentListener, addFocusListener
> > > > addHierarchyBoundsListener, addHierarchyListener > > > >
addInputMethodListener, addKeyListener
> > > > addMouseListener, addMouseMotionListener
> > > > addMouseWheelListener, addPropertyChangeListener > > > >
getComponentListeners, getFocusListeners
> > > > getHierarchyBoundsListeners, getHierarchyListeners > > > >
getInputMethodListeners, getKeyListeners, getListeners > > > >
getMouseListeners, getMouseMotionListeners > > > >
getMouseWheelListeners, getPropertyChangeListeners > > > >
removeComponentListener, removeFocusListener > > > >
removeHierarchyBoundsListener, removeHierarchyListener > > > >
```

```
removeInputMethodListener, removeKeyListener > > > >
removeMouseListener, removeMouseMotionListener > > > >
removeMouseWheelListener, removePropertyChangeListener | | | | |
java.lang.Object
| | | | | java.awt.image.ImageObserver
| | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | java.io.Serializable
| | java.awt.ItemSelectable
> > addItemListener, removeItemListener
| | javax.swing.SwingConstants
| javax.accessibility.Accessible
wsadmin>
```

[5]The primary exception was shown in Table 6-2, where there were more than a dozen listeners for the JTextArea class.

Is it likely that your applications will be using all of these listeners? No, but there are quite a few that are worth investigating. The information in Listing 14-1 shows a large number of listener methods that are part of the java.awt.Component class. The variety and type of listeners in the hierarchy should encourage you, as a developer, and give you confidence that your application will be able to monitor any type of event you need.

Put Your Listener Where Your Component Is

In this section, you learn what it takes to create a MouseListener[6] for a component.[7] Before you begin, though, it is important to realize that if your application is interested in mouse movement, this requires a different kind of listener (MouseMotionListener).[8] For the moment, however, this section focuses on what it takes to create a MouseListener.

Table 14-1 shows the events defined by the MouseListener class. Each of these methods has a MouseEvent parameter that is passed to the event handler method.

[6] See http://docs.oracle.com/javase/8/docs/api/java/awt/event/MouseListener.html.
[7]It's interesting to note that, at least at the time of this writing, the version 7

java.awt.event.MouseListener Javadoc page has a bad link. The “Tutorial: Writing a Mouse Listener” link should actually point to http://docs.oracle.com/javase/tutorial/ uiswing/events/mouselistener.html. This bad link has been corrected on the version 8 page.
[8]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/MouseMotionListener.htı

Table 14-1. *MouseListener Methods*

Method Name	**Invoked When the Mouse** ...
mouseEntered(...)	Enters a component.
mousePressed(...)	Button has been pressed on a component.
mouseReleased(...)	Button has been released on a component.
mouseClicked(...)	Button has been pressed and released on a component.
mouseExited(...)	Exits a component.

Listing 14-2 shows just how easily you can add a MouseListener to a component like a button. Note how the listener class, in lines 10-24, needs to be provided with access to the specific application component, in this case, a JTextArea. This is because it’s external to the Listen1 application class.

Listing 14-2. Adding a MouseListener to a Button

```
10|class listener( MouseListener ) :
11| def __init__( self, textArea ) :
12| self.textArea = textArea
13| def mouseClicked( self, me ) :
14| self.logEvent( me )
15| def mouseEntered( self, me ) :
16| self.logEvent( me )
17| def mouseExited( self, me ) :
18| self.logEvent( me )
19| def mousePressed( self, me ) :
20| self.logEvent( me )
```

```
21| def mouseReleased( self, me ) :
22| self.logEvent( me )
23| def logEvent( self, me ) :
24| self.textArea.append( me.toString() + '\n' )
25|class Listen1( java.lang.Runnable ) :
26| def run( self ) :
27| frame = JFrame(
28| 'Listen1',
29| layout = FlowLayout(),
30| locationRelativeTo = None,
31| size = ( 512, 256 ),
32| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
33| )
34| self.button = frame.add( JButton( 'Button' ) )
35| self.textarea = JTextArea( rows = 10, columns = 40 )
36| self.button.addMouseListener( listener( self.textarea ) )
37| frame.add( JScrollPane( self.textarea ) )
38| frame.setVisible( 1 )
```

Figure 14-1 shows an image from the Listen1.py application. In this case, I moved the mouse over the button, clicked the button, and then moved the mouse elsewhere. Figure 14-1 shows every event that occurred, in the order that they are shown in Table 14-1.

Figure 14-1. *Listen1.py sample output*

What if you aren't interested in monitoring all of the possible events that class provides? Well, one trivial technique is to have the uninteresting methods do nothing (they can contain a simple, pass or return statement). This can be a possible source of confusion and error. I've had at least one instance where I didn't realize that I duplicated a listener method name, where the second instance inadvertently replaced the first. It took me a while to realize and resolve my mistake. What can you do about this kind of situation?

Adapt or Die: Using Adapter Classes

The Swing designers have been very kind to its developers. The hierarchy includes about two dozen "adapter" classes that contain most, if not all, of the methods you'll need, all with empty placeholder statements. Using these as a base class is a great place to start.

One way to find the class you need is by looking at the Javadoc and paying particular interest to the "See Also" sections. For example, the MouseListener Javadoc[6] has, as its first reference in this section, the MouseAdapter class.[9] Using that as a base class for your listener allows you to simplify the listener class from the Listen1 application. Listing 14-2 contains the original listener class in lines 10-24. Listing 14-3 shows a simplified class that assumes you are interested in one only of the available methods.

Listing 14-3. Listen2.py Using MouseAdapter

```
10|class listener( MouseAdapter ) :
11| def __init__( self, textArea ) :
12| self.textArea = textArea
13| def mouseClicked( self, me ) :
14| self.textArea.append( me.toString() + '\n' )
```

Figure 14-2 shows sample output from this iteration of the Listen#.py script. Note that Listen2.py is only interested in mouseClicked events, so you only see references to those events in the text area. You no longer see the other events shown in Figure 14-1.

[9]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/MouseAdapter.html.

Figure 14-2. *Listen2.py sample output*

Listening for Keyboard Events

What if you wanted your application to monitor keystrokes as they are entered, so that the application can be dynamically updated to reflect various information about the user input? If your application has more than one kind of information to be displayed, based on this input, this could make things a little interesting for the developers. One approach would be to have one KeyListener for the user input field that updates multiple fields based on the user's input. One problem with this, though, is that as you continue to add fields to the application, this listener becomes more complex.

Another approach is to have a more generic listener class that is used to monitor one simple property, and then update a single application field based on this property. What would this kind of listener class need in order to perform this role? It would need to have the following parts of the application specified during its instantiation or construction:

• The input field being monitored
• The output field to be updated
• The function used to determine the result displayed in the output field

Say you had a single input field that you wanted to monitor. When the user enters a value, it needs to be checked to see if the value is an even number

(integer). That's not too difficult. What if you also wanted the field to indicate when the value was an odd integer value? It might also be interesting to have a field display whether the value is a prime number.

In order to create this kind of application, you need to understand the kinds of events that are generated for KeyListener objects.[10] How does this class work? Table 14-2 shows the three methods that are part of the KeyListener interface. Each event handler method has a KeyEvent argument that will be passed to the method.

[10]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/KeyListener.html.

Table 14-2. *KeyListener Methods*

Method Name	**Description**
keyPressed(...)	Invoked when a key is pressed.
keyReleased(...)	Invoked when a key is released.
keyTyped(...)	Invoked when a key is typed.

Unfortunately, the description for each method is kind of lacking. What does each mean, and what's the difference between them? When might you be concerned with using one method versus another?
The best way to answer questions like this is to write a program that tries them out. You can then learn more about the methods by seeing the output they produce. Using Listen1.py as a starting point, you can replace MouseListener with KeyListener, replacing each MouseListener method with the appropriate KeyListener method. The result is shown in Listing 14-4.

Listing 14-4. KeyListener Descendant Class from Listen3.py

```
class listener( KeyListener ) :
 def __init__( self, textArea ) :
 self.textArea = textArea
 def keyPressed( self, ke ) :
 self.logEvent( ke )
 def keyReleased( self, ke ) :
```

```
19| self.logEvent( ke )
20| def keyTyped( self, ke ) :
21| self.logEvent( ke )
22| def logEvent( self, ke ) :
23| self.textArea.append( ke.toString() + '\n' )
```

Sample output from the Listen3.py application is shown in Figure 14-3. The first image shows what happens when you press and hold (and finally release) the Right-Shift key. Notice how multiple KEY_PRESSED events, only one KEY_RELEASED event, and no KEY_TYPED events were generated. The second image shows the key events that were generated when you quickly press and release the 0 key. And the third image shows the keystroke events that can occur when a normal key, in this case the 0 key, is pressed and held just long enough to generate two repeated 0’s in the input field.

Listen3
Input:
Clear
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=16,keyText=Shift,keyChar=Un
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=16,keyText=Shift,keyChar=Un
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=16,keyText=Shift,keyChar=Un
java.awt.event.KeyEvent[KEY_RELEASED,keyCode=16,keyText=Shift,keyChar=Ur

Listen3
Input: 0
Clear
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=48,keyText=0,keyChar='0',keyL
java.awt.event.KeyEvent[KEY_TYPED,keyCode=0,keyText=Unknown keyCode: 0x0
java.awt.event.KeyEvent[KEY_RELEASED,keyCode=48,keyText=0,keyChar='0',key

Listen3
Input: 00
Clear
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=48,keyText=0,keyChar='0',k
java.awt.event.KeyEvent[KEY_TYPED,keyCode=0,keyText=Unknown keyCode:
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=48,keyText=0,keyChar='0',k
java.awt.event.KeyEvent[KEY_TYPED,keyCode=0,keyText=Unknown keyCode:
java.awt.event.KeyEvent[KEY_RELEASED,keyCode=48,keyText=0,keyChar='0',

Figure 14-3. *Listen3.py sample output*

From these images, you can see that:

• For repeated keystrokes of characters that don't produce input (such as the Shift key), only KEY_PRESSED events occur.

• For repeated keystrokes of characters that produce input (such as any of the letters and the numeric keys), the KEY_PRESSED and KEY_TYPED events occur in an alternating pattern.

• The KEY_RELEASED event only occurs once for each key.

You are encouraged to use the Listen3.py script and see the events that are generated when you press and release combinations of keys (such as Ctrl-Shift-Spacebar). Are you surprised, or do you observe what you expected? This exercise should help you better understand the keystroke events that are generated.

Listing 14-5 shows almost all of the Listen3 class that was used to produce the output shown in Figure 14-3. At this point, you shouldn't be too surprised by how easy it is to create this kind of application.

Listing 14-5. Listen3 Class from Listen3.py

```
class Listen3( java.lang.Runnable ) :
 def run( self ) :
 frame = JFrame(
 'Listen3',
 layout = FlowLayout(),
 locationRelativeTo = None,
 size = ( 512, 256 ),
 defaultCloseOperation = JFrame.EXIT_ON_CLOSE
 )
 frame.add(
 JLabel(
 'Input:',
 horizontalAlignment = SwingConstants.RIGHT
 )
 )
 self.text = frame.add( JTextField( 8 ) )
 frame.add(
 JButton(
 'Clear',
```

```
43| actionPerformed = self.clear
44| )
45| )
46| self.textArea = JTextArea( rows = 10, columns = 40 )
47| frame.add( JScrollPane( self.textArea ) )
48| self.text.addKeyListener( listener( self.textArea ) )
49| frame.setVisible( 1 )
```

Listing 14-6, on the other hand, may contain a surprise or two. Initially, I intended that the button should only be used to clear the application TextArea. However, after playing with the application for a very short time, I soon realized that it would also be useful to clear the input text field. The potential surprise, however, is related to the last method call in this routine.

What, exactly, does the requestFocusInWindow() method do, and how did I know to use it? Before including this method call, I was forced to use the mouse to click on the input field to give it focus after the button was pressed. The alternative would be to use the Tab key to move the focus from the button to the input field. The problem with this approach is that it generates input that's written to TextArea.

So, I looked for methods in JTextField that had "focus" in their name. The first one that I found was JComponent.requestFocus(),[11] which looked promising, at least until I read that the use of this routine is discouraged. The good news is that the documentation for this method also provided the name of the recommended routine to be used, instead. So, the purpose of the call to the requestFocusInWindow() method is to restore focus to the input field after the button is pressed.

Listing 14-6. The ActionListener Method Used by the Button

```
50| def clear( self, event ) :
51| self.text.setText( " )
52| self.textArea.setText( " )
53| self.text.requestFocusInWindow()
```

All in all, this is a useful application, especially when you are interested in learning about keystroke events that can be generated and the sequence in which they occur.

Most Objects Never Really Listen

At the beginning of this chapter, you read that there are times when it makes sense for multiple listeners to monitor a particular component for the same kind of event. Fortunately, this is something that can easily be accomplished. Before you looked at the kind of keystroke events that can be generated, you saw an application that had multiple listeners monitoring an input field, each one updating a component based on the value contained in the input field.

Before you look at an example, take a moment to think about situations in which it would make sense to do this. Listing 14-7 provides one example. The class constructor for this listener requires that three things be provided:

- The input field being monitored
- The output field to be updated (in this example, a JLabel)
- The function to be used to determine the message value to be specified

Listing 14-7. Listener Class from Listen4.py

```
10|class listener( KeyAdapter ) :
11| def __init__( self, input, msg, fun ) :
12| self.input = input
13| self.msg = msg
14| self.fun = fun
15| def keyReleased( self, ke ) :
16| text = self.input.text
17| if text :
18| try :
19| value = int( self.input.text )
20| msg = [ 'No', 'Yes' ][ self.fun( value ) ]
21| except :
22| msg = 'invalid integer'
23| else :
24| msg = ''
25| self.msg.setText( msg )
```

This listener class, as you can see, descends from the KeyAdapter class,[12] not from the abstract KeyListener class.[10] This allows you to simplify the class because you can choose to implement only the keyReleased(...) method.[13] Maybe you are confused by the syntax shown on line 20. Here, we take

advantage of some of the power of Jython. The first part of the expression—“['No', 'Yes']”—identifies a list containing two values. The second part of the expression—“[self.fun(value)]”—identifies the index used to select the desired value from the list. In order to determine the value to be used, which should be zero or one, you call the user-specified function (provided when the class was instantiated; see lines 11-14) and pass the value of the user-specified input field (after converting it from a string to an integer).

[11]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JComponent.html#requestF

Note In case you are wondering, the try/except statement is necessary in case the user specifies an invalid input and the conversion fails.

Listing 14-8. Listen4 Class from Listen4.py

```
26|class Listen4( java.lang.Runnable ) :
27| def run( self ) :
28| def isEven( num ) :
29| return not ( num & 1 )
30| def isOdd( num ) :
31| return num & 1
32| def isPrime( num ):
33| result = 0 # Default = False
34| if num == abs( int( num ) ) : # Only integers allowed
35| if num == 1 : # Special case
36| pass # use default (false)
37| elif num == 2 : # Special case
38| result = 1 #
39| elif num & 1 : # Only odd numbers...
40| for f in xrange( 3, int( num**0.5 ) + 1, 2 ) :
41| if not num % f :
42| break # f is a factor...
43| else :
44| result = 1 # we found a prime
45| return result
46| def label( text ) :
47| return JLabel(
48| text + ' ',
49| horizontalAlignment = SwingConstants.RIGHT
50| )
```

```
51| frame = JFrame(
52| 'Listen4',
53| layout = GridLayout( 0, 2 ),
54| locationRelativeTo = None,
55| size = ( 200, 128 ),
56| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 57| )
58| frame.add( label( 'Integer:' ) )
59| text = frame.add( JTextField( 10 ) )
60| frame.add( label( 'Even?' ) )
61| even = frame.add( JLabel( '' ) )
62| text.addKeyListener( listener( text, even, isEven ) ) 63| frame.add( label(
'Odd?' ) )
64| odd = frame.add( JLabel( '' ) )
65| text.addKeyListener( listener( text, odd, isOdd ) ) 66| frame.add( label(
'Prime?' ) )
67| prime = frame.add( JLabel( '' ) )
68| text.addKeyListener( listener( text, prime, isPrime ) ) 69| frame.setVisible( 1
)
```

[12] See http://docs.oracle.com/javase/8/docs/api/java/awt/event/KeyAdapter.html.
[13]Initially, I mistakenly used the keyTyped(...) method, thinking that it would be the one invoked after the keystroke had been processed. But I was wrong. Fortunately, it was a trivial thing to change the name of the routine fromkeyTyped(...) to keyReleased(...).

Lines 62, 65, and 68 of Listing 14-8 show how the application has three KeyListener methods for the same input field (the JTextField), as well as specifying different output (JLabel) fields and functions to be called to determine the message to be displayed in each.

Figure 14-4 shows some sample output from the Listen4.py application, which has multiple listeners for a single input field.

Figure 14-4. *Listen4.py output*

Looking for a Listener in a Haystack

Up to this point, things have sort of been handed to you on a platter, so to speak. Unfortunately, that approach is only useful up to a point. Let's take a look at how a developers can determine the best way to do something.

One of the many challenges with designing and developing an application is trying to figure out what size it should be. Along those lines, I wondered how much time and effort would be required to have the application monitor any resize requests that might occur.

Why would I want to do this? Consider, for a moment, that I have no idea how large or small for that matter, to make my application. I could simply let Swing determine the size for me by using the JFrame pack()(inherited) method.

Unfortunately, I haven't always been happy with the default size. So, I did a quick look at the listeners and the Java Swing Tutorial[14] to see if I could find something useful. It didn't take me long to realize that my photographic memory was out of film, and I would have to use a different technique.

Using one of the various Internet search engines, you can search for "java swing resize listener," which will quickly point you to the "How to Write a Component

Listener" swing tutorial page.[15] Looking at the sample application on that page, called ComponentEventDemo.java,[16] I was able to find some things worth investigating.

• This application, like most from the Java Swing Tutorial pages, has the main(...) method use an anonymous Runnable class within its call to the SwingUtilities invokeLater(...) method.

• Invariably, the run(...) method in this class simply calls the createAndShowGUI(...) method within the application class.

• This application class (ComponentEventDemo) extends JPanel, so the createAndShowGUI(...) method creates JFrame instances and uses the JPanel that is created by the application class as a replacement for the JFrame ContentPane. I haven't used this technique, but it appears to be a common one used by many of the sample applications in the Swing Tutorial pages.

• The sample applications that extend a base class like JPanel almost always use the Java technique of calling the super(...) method to invoke the base class constructor. Unfortunately, this isn't available to Jython developers, so instead, you'll have to invoke the base class constructor—that is, call the JPanel.__init__(self) method.

• The only expression that can't be used "as-is" occurs in the component event handler methods, and looks like this: e.getComponent().getClass().getName(). If you try to use expressions like these in your Jython scripts, you'll get an exception something like this:

TypeError: getName(): expected 1 args; got 0 [14]See https://docs.oracle.com/javase/tutorial/uiswing/. [15]See http://docs.oracle.com/javase/tutorial/uiswing/events/componentlistener.html. [16]See http://docs.oracle.com/javase/tutorial/uiswing/examples/events/ComponentEvent ComponentEventDemo.java.
The reason that this happens is because Jython has two ways for calling a method instance[17]:

• theInstance.theMethod(args)
• TheClass.theMethod(theInstance, args)

Unfortunately, you are trying to use the first technique, but Jython prefers the second. How can you fix this? You have a couple of different options available. You can use the syntax preferred by Jython, which is an expression like this one:

```
java.lang.Class.getName( e.getComponent().getClass() )
```

Unfortunately, this results in a string something like " javax.swing.JButton", instead of the preferred "JButton". A somewhat simpler approach is to use the fact that you can use str(e.getComponent().getClass()) to get this same string value, then split the string with ". " as a delimiter. Finally, you use the negative list indexing technique to access the final part of the list of strings:

```
name = str( e.getComponent().getClass() ).split( '.' )[ -1 ]
```

Before you take a look at Jython script that I produced, try to convert the original Java application to Jython yourself. This exercise should provide you with some practice at reading an original Java Swing application and trying to produce a script using the same kind of components and organization.

Using the original Java version—or the one you created, or even the one that can be found in code\Chap_14\ ComponentEventDemo.py—take a look at the events generated when you do something simple like make the application window a tiny bit taller. Figure 14-5 shows multiple images produced by using the Jython script at various times. The most interesting one, at least to me, is the last image, which shows just how many "Moved" events are generated when you make the window a little bit larger.[18] Also interesting is the fact that only one "Resized" event is generated. The really interesting part is that the event identifies the component that was resized, in this case, the JFrame object.

[17]Thanks to Jeff Emanuel for providing this explanation on the jython-users mailing list.

[18]This is why my version of the script includes a counter showing the number of events that have been encountered.

ComponentEventDemo
1: JFrame --- Resized
2: JPanel --- Resized
3: JPanel --- Resized
4: JPanel --- Moved
5: JCheckBox --- Resized
6: JCheckBox --- Resized
7: JCheckBox --- Moved
8: JButton --- Resized
9: JFrame --- Resized
10: JFrame --- Moved
11: JFrame --- Shown
Clear
Button visible

ComponentEventDemo
12: JButton --- Hidden
13: JPanel --- Resized
14: JPanel --- Moved
15: JCheckBox --- Moved
Button visible

Figure 14-5. *ComponentEventDemo sample output*

Using a ComponentAdapter to Monitor Changes

Using this ComponentEventDemo application made me think about using a ComponentListener[15] to gain a better understanding about the size and position of an application window. Of course, if you aren't interested in monitoring all of the Component events, you might want to consider using a ComponentAdapter[19] instead.

What do you think it would take to have an application that displays the width and height of the frame? The images in Figure 14-6 show one possible representation. Take a few moments and see if you can figure out how to do this before looking at the source, which is available in code\Chap_14\Frame1.py.

Figure 14-6. *Frame1 sample output*

Issues you might want to consider:
• Which Layout Manager do you want to use?
• For which events does the application need to listen?

Listing 14-9 shows the Frame1 class from the script of the same name. Notice how the Layout Manager attribute of the frame constructor is initialized to None. That means that this application is going to position each component by specifying its size and location.

Listing 14-9. Frame1 Class from Frame1.py

```
16|class Frame1( java.lang.Runnable ) :
17| def run( self ) :
18| self.frame = frame = JFrame(
19| 'Frame1',
20| size = ( 200, 200 ),
21| layout = None,
22| locationRelativeTo = None,
23| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
24| )
25| frame.addComponentListener( listener( self ) )
26| insets = frame.getInsets()
27| self.width = JTextField( 4 )
28| self.height = JTextField( 4 )
29| items = [
30| [ JLabel( 'Width:' ), 7, 7 ], 31| [ self.width , 50, 5 ], 32| [ JLabel( 'Height:' ), 7,
31 ], 33| [ self.height , 50, 30 ] 34| ]
35| for item in items :
```

```
36| thing = frame.add( item[ 0 ] ) 37| size = thing.getPreferredSize() 38|
thing.setBounds(
39| insets.left + item[ 1 ], 40| insets.top + item[ 2 ], 41| size.width,
42| size.height
43| )
44| frame.setVisible( 1 )
```

[19]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/ComponentAdapter.html.

Listing 14-10 shows the simple listener class that is a descendant of the abstract ComponentAdapter class. This allows you to simplify your class and only specify methods of interest, which in this case is the componentResized() method.[20]

Listing 14-10. Listener Class from Frame1.py

```
8|class listener( ComponentAdapter ) : 9| def __init__( self, app ) :
10| self.app = app
11| def componentResized( self, ce ) : 12| app = self.app
13| size = app.frame.getSize() 14| app.width.setText( `size.width` ) 15|
app.height.setText( `size.height` )
```

What would it take to have an application that shows the position or location on the screen, in addition to the size? It would be a good exercise for you to try to expand Frame1.py to add this kind of functionality. Why don't you take some time to do this before you take a look at the one found in code\Chap_14\Frame2.py?

Listing 14-11 shows the revised listener class that was used to generate the output shown in Figure 14-7. Note how additional ComponentListener methods have been included. This allows the application to update the position, that is the X and Y fields, as the application frame is moved. In order to do this, the frame.getBounds() method is called to acquire the current size and position of the frame. In Listing 14-8, you were only using the size of the application, so the frame.getSize() method was used instead.

[20]If you are unfamiliar with the backtic operator, it converts the expression within as a printable string. See the Jython repr() built-in function.

Listing 14-11. Listener Class from Frame2.py

8|class listener(ComponentAdapter) : 9| def __init__(self, app) :
10| self.app = app
11| def updateInfo(self) :
12| app = self.app
13| bounds = app.frame.getBounds() 14| app.width.setText(`bounds.width`) 15| app.height.setText(`bounds.height`) 16| app.x.setText(`bounds.x`)
17| app.y.setText(`bounds.y`)
18| def componentMoved(self, ce) : 19| self.updateInfo()
20| def componentResized(self, ce) : 21| self.updateInfo()
22| def componentShown(self, ce) : 23| self.updateInfo()

Figure 14-7. *Frame2 sample output*

Listing 14-12 shows just how little needed to be modified in order to have the application monitor the location as well as the size. It also shows how tedious it can be to determine the components' positions so that they can be identified without using a Layout Manager.

Why did I choose to use the absolute layout technique? The primary reason was because I didn't want the components to move around on the application as it moved and was resized.

Listing 14-12. Modified Statements from the Frame2 Class

24|class Frame2(java.lang.Runnable) :
| ...
34| self.width = JTextField(4)
35| self.height = JTextField(4)
36| self.x = JTextField(4)

```
37| self.y = JTextField( 4 )
38| items = [
39| [ JLabel( 'Width:' ) , 11, 7 ],
40| [ self.width , 50, 5 ],
41| [ JLabel( 'Height:' ), 7, 31 ],
42| [ self.height , 50, 30 ], 43| [ JLabel( 'X:' ) , 35, 55 ], 44| [ self.x , 50, 53 ], 45| [
JLabel( 'Y:' ) , 35, 79 ], 46| [ self.y , 50, 78 ] 47| ]
| ...
```

Monitoring the Input Fields

Using the mouse to move and resize the application window can be a frustrating experience, especially if you are trying to precisely adjust a specific value (such as the height). Wouldn't it be useful to also allow users to enter a value in any of these input fields as well? If you do allow user input, you'll need to verify that input before it is used. For example, it doesn't make any sense to allow users to specify a negative width or height. On the other hand, it may make sense for the location values to be negative. On my system, I have a second display, and when I put the application frame in the middle of it, the value of X is -775.

So how do you go about monitoring the input fields? One way is to have an ActionListener for each field. In fact, you could have a different one for each field so that you don't have to figure out which value was modified. Listing 14-13, which can be found in Frame3.py, shows one way that this can be done.

Listing 14-13. Unique ActionListener for Each Input Field

```
24|class Frame3( java.lang.Runnable ) :
25| def changeWidth( self, event ) :
26| value = event.getActionCommand()
27| try :
28| width = int( value )
29| size = self.frame.getSize()
30| self.frame.setSize( width, size.height )
31| except :
32| print 'Invalid Width: "%s"' % value | ...
57| def run( self ) :
| ...
66| self.width = JTextField(
```

```
67| 4, actionPerformed = self.changeWidth
68| )
69| self.height = JTextField(
70| 4, actionPerformed = self.changeHeight
71| )
72| self.x = JTextField(
73| 4, actionPerformed = self.changeX
74| )
75| self.y = JTextField(
76| 4, actionPerformed = self.changeY
77| )
| ...
```

Unfortunately, this isn't a great approach for a number of reasons, including these:

• Using ActionListeners means that the values aren't verified until the user presses Enter, which isn't very user friendly. It would probably be better if verification occurred when the user does one of the following:

• Presses the Enter key
• Tries to change focus (using the Tab key, Shift-Tab, or even a mouse event)
• Enters the text

• Each ActionListener is very similar. You might be able to refactor, but the technique used to indicate that an invalid value was specified isn't very good or user friendly. Nor does it use the Swing interface, which is a bad thing.

Back in Chapter 7, you read a little about the JFormattedTextField class,[21] including how it is used to restrict user input. Let's revisit that class and see if it can help you improve the user experience of this application. What part of a JFormattedTextField will help you with this application? Well, there are some things in the Javadoc that you should be aware of. Specifically, it contains information about using one of the following:

• A PropertyChangeListener[22] for monitoring editing changes
• An InputVerifier[23] to keep focus from being lost when an invalid value is specified

Using a PropertyChangeListener

One of the interesting things about the JFormattedTextField class is that, in addition to the text attribute that it inherits from JTextField, it also defines a value attribute.[24] Why does it have both a text and a value attribute? More importantly, what does that mean for your applications?

Consider the following scenario: a formatted text field has some kind of value, for example, some kind of currency. When this field has focus, it may be an indication that the user wants to modify the value, which is great. Users start typing and as they do so, the text that they enter might not be valid. As user input is being provided, the text attribute reflects what the user has entered. When entry is complete, the contents of the text attribute can be used to determine if the information is valid or appropriate for the field.

So, the value attribute maintains a valid value, formatted using the appropriate pattern. The text attribute, on the other hand, reflects the current, unverified, and unformatted user input. It's when the user entry process is complete that the verification and formatting can occur, but only if the user-specified value is acceptable. At least that way it is supposed to work, if your applications use the available capabilities.

Let's start by looking at the PropertyChangeListener to see how it fits into this strategy. To begin, consider an application that has a couple of formatted text fields with a property change listener event handler routine to tell you about the PropertyChangeEvents that occur. This routine can update a text area with information about the events, including the PropertyName, the OldValue, and the NewValue attributes identified by PropertyChangeEvent.[25]

Listing 14-14 shows the PropertyChangeListener and ActionListener event handler routines from the PropertyListener.py sample application.

[21] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFormattedTextField.html.
[22]See http://docs.oracle.com/javase/8/docs/api/java/beans/PropertyChangeListener.html
[23]See http://docs.oracle.com/javase/8/docs/api/javax/swing/InputVerifier.html.
[24]In spite of the fact that the Javadoc for this class doesn't identify this fact, at least not explicitly in the "Field Summary" section.
[25]See

http://docs.oracle.com/javase/8/docs/api/java/beans/PropertyChangeEvent.html.

Listing 14-14. PropertyChangeListener Event Handler Routine

```
class PropertyListener( java.lang.Runnable ) :
 def changed( self, pce ) :
 format = ' Name: %(name)s\n'
 format += 'OldValue: %(old)s\n'
 format += 'NewValue: %(new)s\n\n'
 name = pce.getPropertyName()
 old = 'pce.getOldValue()'
 new = 'pce.getNewValue()'
 self.textArea.append( format % locals() )
 def clear( self, e ) :
 self.textArea.setText( '' )
```

Figure 14-8 shows some images from this application. I’m not sure what I was expecting, but I didn’t realize the kind of PropertyChangeEvents that would be generated (which is exactly why this kind of script is so useful).

Figure 14-8. *PropertyListener.py sample output*

Figure 14-8. (*continued*)

The images in Figure 14-8 demonstrate various events based on the user input. Arguably, the most important is when the user enters a valid value for the formatted text field. This property change event will have a property name of value, as well as an OldValue attribute that can be used to determine the previous

valid value, and a NewValue attribute that can be used to identify the newly entered value. This is shown in the last image in Figure 14-8.

Using an InputVerifier

As mentioned earlier, the InputVerifier class exists to allow your applications to check for valid input before field focus is enabled. Let's take a look at a trivial application that uses an InputVerifier to force the user to enter the value pass before the input focus can change. Figure 14-9 shows the output of this simple application.

Figure 14-9. *VerifyTest1.py sample output*

Listing 14-15 shows how the InputVerifier only needs to implement the verify(...) method. As long as this verify(...) method returns a value of false (0), the input focus isn't allowed to change.

Listing 14-15. Trivial Input Verification from VerifyTest1.py

```
9|class inputChecker( InputVerifier ) :
10| def verify( self, input ) :
11| return input.getText() == "pass"
12|class VerifierTest1( java.lang.Runnable ) :
13| def run( self ) :
14| frame = JFrame(
15| 'VerifierTest1',
16| locationRelativeTo = None,
17| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
18| )
19| frame.add(
20| JTextField(
21| 'Enter "pass"',
22| inputVerifier = inputChecker()
23| ),
24| BorderLayout.NORTH
25| )
```

```
26| frame.add(
27| JTextField( 'TextField 2' ),
28| BorderLayout.SOUTH
29| )
30| frame.pack()
31| frame.setVisible( 1 )
```

The problem with this trivial implementation is that the user doesn't get much feedback that the current value is unacceptable. Wouldn't it be better if you used some of the available techniques to explain to the user that something is wrong?

Figure 14-10 shows one possible approach. Notice what happens when the user simply tries to change focus by pressing the Tab key. The border color around the input field changes to RED and the field text is selected. I don't know about you, but I find this to be a pretty strong indication that something is wrong here.

Figure 14-10. *VerifyTest2.py sample output*

Listing 14-16 shows this modified InputVerifier descendant class. In it, the verify(...) method does a little more than the previous version. This additional functionality requires fewer than a dozen lines of code. However, there is a world of difference between the usability of the two applications, don't you think?

Listing 14-16. A Slightly More Involved Input Verifier from VerifyTest2.py

```
11|class inputChecker( InputVerifier ) :
12| def __init__( self ) :
13| self.border = None # holder for "original" border
14| def verify( self, input ) :
15| text = input.getText()
16| if not self.border : # The first time, save the border
17| self.border = input.getBorder()
18| result = input.getText() == "pass"
```

```
19| if result : # valid? Restore original border
20| input.setBorder( self.border )
21| else : # invalid? change border color
22| input.setBorder( LineBorder( Color.red ) )
23| input.selectAll()
24| return result
```

Is there anything missing from this application? Consider what happens when an invalid value is entered. Nothing, that's what. The InputVerifier isn't invoked because it is associated with events that attempt to change the keyboard focus. Pressing the Enter key doesn't do that. This event is associated with ActionEvents, not with InputVerifier events.

Figure 14-11 shows what happens when you associate an ActionListener with the first input field that invokes the InputVerifier to validate the user input. The second image shows that, when the user presses the Enter key without changing the text, the InputVerifier indicates that the text is invalid. The final image shows what happens when the user enters valid text and presses Enter. Note that focus remains with the input field because the change of focus is not a normal result of this action.

Figure 14-11. *VerifyTest3.py sample output*

Listing 14-17 shows how simple it can be to have the ActionListener event handler routine invoke the InputVerifier shouldYieldFocus() method to initiate input verification.

Listing 14-17. Using an ActionListener Method to Verify Input from VerifyTest3.py

```
25|class VerifierTest3( java.lang.Runnable ) :
26| def verify( self, e ) :
27| self.verifier.shouldYieldFocus( e.getSource() )
28| def run( self ) :
| ...
34| self.verifier = inputChecker()
```

```
35| frame.add(
36| JTextField(
37| 'Enter "pass"',
38| actionPerformed = self.verify,
39| inputVerifier = self.verifier
40| ),
41| BorderLayout.NORTH
42| )
| ...
```

Summary

This chapter covered some of events that your Swing applications will encounter. It is important to be familiar with these events so that your applications can react well to user input. Remember, though, that there are many, many events in the Swing class hierarchy. This chapter provides exposure to some of them.

In Chapter 15, you'll investigate how to use open source software libraries to enhance your applications. **Chapter 15**

Nuts to Soup: Using Jsoup to Enhance Applications

Up to this point, the applications in this book have been pretty simple. In this chapter, you learn what it will take to create an application "as you go." By that, I mean you can start simple and iterate over its creation and development, with the intent of making it more useful based on what you learn as you go along. Additionally, you will use an existing open source software (OSS) Java class library to create a useful application built on the work of others.

Using Existing Classes: Creating an HTML Retrieval Application from Scratch

Based on the number of times that I've referred to the Java Swing documentation pages (the Javadoc for the Swing class hierarchy), along with the number of footnote references that point to various class pages, it would be useful if you had an application to help you with this kind of lookup, don't you think? So, what will such an application need to do?

Wouldn't it be great if this application were able to access and process the Javadoc pages directly? What would that take to achieve? This application needs to be able to access a remote website, request a specific page, and process the results.

Are there any classes in the Swing hierarchy that you might be able to use? Listing 15-1 shows the list of classes on the Java "All Classes" page[1] that include HTML somewhere in their class name.[2]

Listing 15-1. Java "HTML" Classes

```
BasicHTML
HTML
HTML.Attribute
HTML.Tag
HTML.UnknownTag
HTMLDocument
HTMLDocument.Iterator
HTMLEditorKit
HTMLEditorKit.HTMLFactory HTMLEditorKit.HTMLTextAction
HTMLEditorKit.InsertHTMLTextAction HTMLEditorKit.LinkController
HTMLEditorKit.Parser
HTMLEditorKit.ParserCallback HTMLFrameHyperlinkEvent
HTMLWriter
MinimalHTMLWriter
```

[1] See http://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html.

[2] This list of classes can be obtained by executing the following command: jython getLinks.py http://docs.oracle.com/ javase/8/docs/api/allclasses-noframe.html | grep HTML. Then edit the result to remove the URLs that are listed with each class. The getLinks.py script is discussed later in this chapter.

This would appear to be a reasonable place to start an investigation. How could you use some of these classes in order to:

- Connect to a remote website using a uniform resource locator (URL)
- Retrieve the hypertext markup language (HTML) from that page
- Process the contents, with the purpose of locating the hyperlinks[3] and the associated text

After a bit of investigation and work, I was able to get the routine shown in Listing 15-2 working. There may be some way to refactor the code to make it slightly shorter, but I don't see any easy way to make it easier to read and understand.

Listing 15-2. The getLinks Routine

```
def getLinks( page ) :
 url = URI( page ).toURL()
 conn = url.openConnection()

 isr = InputStreamReader( conn.getInputStream() )
 br = BufferedReader( isr )
 kit = HTMLEditorKit()
 doc = kit.createDefaultDocument()
 parser = ParserDelegator()
 callback = doc.getReader( 0 )
 parser.parse( br, callback, 1 )
 iterator = doc.getIterator( HTML.Tag.A )
 while iterator.isValid() :
 try :
 attr = iterator.getAttributes()
 src = attr.getAttribute( HTML.Attribute.HREF )
 start = iterator.getStartOffset()
 fini = iterator.getEndOffset()
 length = fini - start
 text = doc.getText( start, length )
 print '%40s -> %s' % ( text, src )
 except :
 pass
 iterator.next()
```

What does the output of the script containing this routine look like? Well, Listing 15-3 shows the (slightly massaged) output of the script, which allows the output to fit in the available space. This figure shows the hyperlinks found on the IBM website (http://www.ibm.com).[4]

[3]See http://en.wikipedia.org/wiki/Hyperlink.

Listing 15-3. Absolute Links from http://www.ibm.com

United States -> http://www.ibm.com/planetwide/select/selector.html IBM?? ->

http://www.ibm.com/us/en/
Site map -> http://www.ibm.com/sitemap/us/en/
Figure 15-1 shows output from the Merriam Webster page. It contains a number of relative hyperlinks instead of the absolute links shown in Listing 15-3.

```
                -> /
    "Cartouche" -> /dictionary/cartouche
   "Praemunire" -> /dictionary/praemunire
   "Positivity" -> /dictionary/positivity
Quizzes & Games -> /game/index.htm
Word of the Day -> /word-of-the-day/
          Video -> /video/index.php
      New Words -> http://nws.merriam-webster.com/opendictionary/
   My Favorites -> /my-saved-words/manage-list.htm
```

Figure 15-1. *Relative links from http://www.merriam-webster.com/*

Dealing with relative versus absolute hyperlinks is just the kind of thing that your application should take into consideration. Additionally, dealing with HTML includes the risk that it may not be well formed (i.e., syntactically correct).

Based on the hope that others are likely to have encountered and already solved these kinds of problems, I decided to search the Internet for possible solutions. An initial search for “Jython HTML parsing” results in some interesting possibilities. Unfortunately, subsequent investigation identifies some prohibiting factors, including the fact that some Python modules (such as urllib2, HTMLParser, BeautifulSoup, and lxml) don’t work well or at all with Jython environments or the possible solutions have significant performance issues.

Wouldn’t It Be Nice: Using Java Libraries

If the Jython or Python modules can’t help , what’s left? Don’t forget that one of the most significant advantages to Jython is that it is running on a Java Virtual Machine. So, you can investigate the possibility of using one or more Java libraries to help.

[4]This output is the result of executing wsadmin -f %SwJCode%\Chap_15\getLinks.py -conntype none with the SwJCode environment variable containing the path to the code folder that contains the book’s scripts.

When I asked on the Jython user's mailing list[5] for suggestions, the *Jsoup*[6] *library* was quickly recommended. Jsoup is a Java library for working with "real-world" HTML (such as pages with syntax issues). The Jsoup page includes the following quote:

Note Jsoup is a Java library for working with real-world htMl. it provides a very convenient api for extracting and manipulating data, using the best of DoM, Css, and jQuery-like methods.

Jsoup implements the WhatWg htMl5[7] specification and parses htMl to the same DoM as modern browsers do. This sounds exactly like the kind of thing you need in this case, don't you think?

Working with the Jsoup Library

Follow these steps to use a new library with wsadmin and Jython:
1. First, you need to download the Jsoup archive library:

• Point your favorite browser to http://jsoup.org/.
• Select the Download link.
• Right-click on the core library file[8] and save it in a convenient directory.

2. For wsadmin, you need to add this Java archive to the classpath. Fortunately, a wsadmin command-line option exists to do exactly this. If the Jsoup JAR file is in the C:\temp directory, this can be as simple as:[9]

```
wsadmin -wsadmin_classpath c:\temp\jsoup-1.8.1.jar -f scriptName.py
```

For Jython, you only need to have this library (JAR file) as part of the JYTHONPATH environment variable. For example:

```
Set JYTHONPATH=C:\Programs\jsoup\jsoup-1.8.1.jar;%JYTHONPATH%
jython scriptName.py
```

[5] See jython-users@lists.sourceforge.net.
[6] See http://jsoup.org/.
[7] See http://whatwg.org/html.
[8] At the time of this writing, it was jsoup-1.8.1.jar.
[9] The first time that Jython is told about this Java archive, you will see a message something like this to indicate that Jython is aware of the archive and has processed it to store information about the JAR file for later use by Jython

scripts.

sys-package-mgr: processing new jar, 'C:\temp\jsoup-1.8.1.jar'.

Listing 15-4 shows a simple, interactive wsadmin session that demonstrates how easy it is for scripts to use the Jsoup library.
Listing 15-4. Simple Jsoup Demonstration

```
1|wsadmin>from org.jsoup import Jsoup
2|wsadmin>
3|wsadmin>doc = Jsoup.connect( 'http://www.ibm.com' ).get()
4|wsadmin>doc.title()
5|'IBM - United States'
6|wsadmin>for link in doc.getElementsByTag( 'a' ) :
7|wsadmin> print link.attr( 'abs:href' )
8|wsadmin>
9|http://www.ibm.com/planetwide/select/selector.html
10|http://www.ibm.com/us/en/
11|http://www.ibm.com/sitemap/us/en/
12|http://www.ibm.com/software/marketing-solutions/benchmark-hub/...
13|http://www.ibm.com/systems/infrastructure/us/en/it-infrastruct...
14|http://www-935.ibm.com/services/us/en/it-services/business-con...
15|http://www.ibm.com/common/twitter/ibm.xml
16|http://www.ibm.com/news/us/en
17|...
```

Table 15-1 contains a description of each line of the simple interactive session.

Table 15-1. *Simple Jsoup Demonstration, Explained* **Lines Description**
1 Import statement used to add the Jsoup library to the current namespace. 2 An empty line to make the session input easier to read.

3 A statement used to retrieve the contents of the www.ibm.com web page.
Note: It is possible for an exception to be raised.
4 Demonstration of using the title() method[10] to display the title of the retrieved web page.
6-8 For loop used to retrieve the HTML links and display the absolute URLs[11] to which they refer.
9. . . The remainder of the session shows the first few lines that are output by the

previous loop.

Isn't this easier to read, understand, and use than the ones from the Swing hierarchy? I think so, and I'm pretty sure that you'll agree.
[10]See http://jsoup.org/apidocs/org/jsoup/nodes/Document.html#title(). [11]See http://jsoup.org/cookbook/extracting-data/working-with-urls.

Jsoup Call May Appear to Hang

So now you're all set, right? No, not quite. Remember that the Swing applications need to be able to deal with events. More importantly, you don't want the application to wait or appear to be "hung," while waiting for things (such as like the retrieval of web pages) to complete. As mentioned before, you're going to start simple and make decisions based on how things look.

First you need a descendant of the SwingWorker class to perform the Jsoup retrieval and to process requests on a separate thread. Listing 15-5 shows the soupTask class from this simple application.
Listing 15-5. soupTask Class from javadocInfo_01.py

```
23|class soupTask( SwingWorker ) :
24| def __init__( self, comboBox, label, url ) :
25| self.cb = comboBox # Save provided references
26| self.msg = label
27| self.url = url # URL to be used
28| self.doc = None
29| SwingWorker.__init__( self )
30| def doInBackground( self ) :
31| try :
32| self.msg.setText( 'working...' )
33| self.doc = Jsoup.connect( self.url ).get()
34| self.msg.setText( 'ready' )
35| except :
36| Type, value = sys.exc_info()[ :2 ]
37| print 'Error:', str( Type )
38| print 'value:', str( value )
39| self.msg.setText( str( value ) )
40| if self.doc :
41| self.cb.removeAllItems()
```

```
for link in self.doc.getElementsByTag( 'a' ) :
self.cb.addItem( str( link.text() ) )
def done( self ) :
pass
```

Figure 15-2 shows some sample images from the first Swing application using the Jsoup library.

Figure 15-2. *javadocInfo_01.py sample output*

The last image in Figure 15-2 shows that a browser appears when the same URL is used. It certainly looks like the Jsoup application is working as expected.

From a Combo Box to a List Box

That application is quite simple and demonstrates the feasibility of using the Jsoup library to process the current Java documentation URL.[12] Unfortunately, a combo box doesn't appear to be a good choice for showing the list of available entries, does it? What else is missing? Well, I would like to be able to see the URL of the selected entry, wouldn't you? Let's take a look at how to do that.

Figure 15-3 shows some images from this iteration of the application. The first image shows the application as a connection to the remote host is being established. The second image, which is very short lived, shows that the processing of the retrieved information is in progress. Then you see an image of the list of items from the page, and finally, you see that when you make a selection, the text field is updated to reflect the associated URL for the selection.

Figure 15-3. *javadocInfo_02.py sample output*

To make this happen, you have to change the JComboBox into a JList, which allows you to display more entries. This means that you need to change the soupTask class to deal with these changes. Additionally, a new argument is added to this constructor, so that the task can return a dictionary of the links found on the page, indexed by the text associated with each link. Listing 15-6 shows the most significant changes that were made to the application.

[12]See http://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html.

Listing 15-6. soupTask, Iteration 2, from javadocInfo_02.py

```
class soupTask( SwingWorker ) :
```

```
27| def __init__( self, List, label, url, result ) :
28| self.List = List # Save provided references
29| self.msg = label
30| self.url = url # URL to be used
31| self.doc = None
32| self.docLinks = result # lookup result
33| SwingWorker.__init__( self )
34| def doInBackground( self ) :
35| try :
36| self.msg.setText( 'connecting...' )
37| self.doc = Jsoup.connect( self.url ).get()
38| self.msg.setText( 'processing...' )
39| model = DefaultListModel()
40| for link in self.doc.getElementsByTag( 'a' ) :
41| name = link.text()
42| href = link.attr( 'abs:href' )
43| self.docLinks[ name ] = href
44| model.addElement( name )
45| self.List.setModel( model )
46| self.msg.setText( 'Make a selection' )
47| except :
48| Type, value = sys.exc_info()[ :2 ]
49| Type, value = str( Type ), str( value )
50| print 'Error:', Type
51| print 'value:', value
52| self.msg.setText( value )
53| def done( self ) :
54| pass
```

That’s pretty good, but it’s still not a great application. What else can you do to make it more useful and usable? Listing 15-7 shows the main application class of the second iteration of the Javadoc lookup application. Overall, it’s amazing how much this little application can do with fewer than 100 lines of Jython code.

Listing 15-7. javadocInfo_02 Application Class

```
55|class javadocInfo_02( java.lang.Runnable ) :
56| def run( self ) :
57| frame = JFrame(
58| 'javadocInfo_02',
```

```
59| locationRelativeTo = None,
60| size = ( 350, 225 ),
61| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
62| )
63| model = DefaultListModel()
64| model.addElement( 'One moment please...' )
65| self.List = JList(
66| model,
67| valueChanged = self.pick,
68| selectionMode = ListSelectionModel.SINGLE_SELECTION
69| )
70| frame.add(
71| JScrollPane( self.List ),
72| BorderLayout.CENTER
73| )
74| self.msg = JTextField()
75| frame.add( self.msg, BorderLayout.SOUTH )
76| self.Links = {}
77| soupTask(
78| self.List, # The visible JList instance 79| self.msg, # The message area
(JTextField) 80| JAVADOC_URL, # Remote web page URL to be processed 81|
self.Links # Dictionary of links found 82| ).execute()
83| frame.setVisible( 1 )
```

Listing 15-8 shows the ListSelectionListener[13] event handler that's invoked when the user makes a selection. It takes a bit of work (code) to figure out when this handler actually has some work to do. Why is that? One reason is that it is invoked when a ListSelectionEvent[14] occurs. So, the first thing it does is determine if the event that occurred is related to a selection adjustment, which is ignored. Then it checks the number of entries on the list. If only one element is present, which only occurs when the list is created with the "One moment please. . ." message, this event is ignored.

Finally, it checks to see if an element is selected, in which case the URL associated with this item is displayed in the message text field. Otherwise, the "Make a selection" message is displayed.

Listing 15-8. javadocInfo_02.py ListSelectionListener Event Handler

```
84| def pick( self, e ) :
```

```
if not e.getValueIsAdjusting() :
List = self.List
model = List.getModel()
if model.getSize() > 1 :
index = List.getSelectedIndex()
if index > -1 :
choice = model.elementAt( index )
self.msg.setText( self.Links[ choice ] )
else :
self.msg.setText( 'Make a selection' )
```

Adding a TextArea to Show the HTML

From here, it's easy to imagine how useful it would be to retrieve the contents of the page that is selected by the user. You really don't need to create another web browser.[15] It is potentially useful if the application can retrieve, process, and analyze a user-selectable page. Let's begin by displaying the page's contents (the HTML), and then decide how best to proceed. Listing 15-9 contains the modified textTask class, which is where the most interesting changes occur.

[13] See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/ListSelectionListener.
[14]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/ListSelectionEvent.ht
[15]In fact, if you try to display a complete HTML page using one of the HTML-aware components, you are likely to encounter a large number of Java exceptions.

Listing 15-9. textTask Class from javadocInfo_03.py

```
class textTask( SwingWorker ) :
def __init__( self, area, url ) :
self.area = area # Save area to be updated
self.url = url # URL to be retrieved
SwingWorker.__init__( self )
def doInBackground( self ) :
try :
self.area.setText( 'connecting...' )
```

```
68| doc = Jsoup.connect( self.url ).get()
69| self.area.setText( str( doc.normalise() ) )
70| except :
71| Type, value = sys.exc_info()[ :2 ]
72| Type, value = str( Type ), str( value )
73| self.area.setText(
74| '\nError: %s\nValue: %s' % ( Type, value )
75| )
76| def done( self ) :
77| pass
```

Figure 15-4 shows some sample output from the javadocInfo_03.py script. The first image shows the HTML text from the JFrame Javadoc web page. It should be no surprise that the TextArea containing the HTML needs to be in a scroll pane. The second image shows the portion of the HTML that describes the Field Summary portion of the page. The last image shows how this HTML is displayed in a browser.

Figure 15-4. *javadocInfo_03 sample output*

Figure 15-4. (*continued*)

Rendering HTML

It certainly would be nice if you could actually use HTML tags to tell how the Swing component should display information. Well, you can, at least to a limited extent. Some, but not all, Swing components are capable of using HTML tags to determine how information should be displayed. You saw from the most recent application output in Figure 15-4 that the JTextArea component doesn't do this.

Don't get your hopes up too quickly. You won't be able to create a fully capable web browser in your applications simply by selecting the "right" Swing components.
Another caveat you need to be aware of is the fact that the HTML specification

implemented in the Swing component hierarchy is 3.2, so it’s not as feature-rich as the latest HTML specification (version 5), which is starting to show up on the Internet.

What does it take to have one of the HTML capable components display text as the HTML tags indicate? Well, it’s as simple as having the text attribute string begin with <html>.

Listing 15-10 contains the trivial script that demonstrates just how easy it is to use HTML to do this. Please be careful though—just because you can do something, doesn’t mean that you should.

Listing 15-10. HTMLtext_01.py Script Showing an HTML Label

```
1|from javax.swing import JFrame
2|from javax.swing import JLabel
3|frame = JFrame(
4| 'HTMLtext_01',
5| size = ( 200, 200 ),
6| locationRelativeTo = None,
7| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
8| )
9|text = '<html>'
10|text += '<sup>My</sup> '
11|text += '<sub><i>Label</i></sub> '
12|text += '<font color="#FF0000">is</font> '
13|text += '<font color="#00FF00"><b>far</b></font> ' 14|text += '<font
color="#0000FF">too busy,</font> ' 15|text += "<u>isn't it?</u>"
16|label = frame.add( JLabel( text ) )
17|frame.setVisible( 1 )
18|raw_input()
```

Figure 15-5 demonstrates this axiom quite vividly. Keep in mind that you should be interested in making your application more appealing, not less.

Figure 15-5. *HTMLtext_01.py sample output*

If you are familiar with HTML, you know that you can do all sorts of interesting things with it. For example, as you can see in Figure 15-6, you can create a simple unordered list tag (starts on line 10 and ends on line 14 of Listing 15-?) to display a bulleted list with each item in a different color.

Figure 15-6. *HTMLtext_02.py sample output*

Listing 15-11 shows the trivial script used to produce the output in Figure 15-6. As you can see, it’s up to the developer how to build the string. In this case, using multiple concatenation steps to build the HTML string makes for a better

and more complete understanding of the various parts of the HTML.

Listing 15-11. HTMLtext_02.py Source

```
from javax.swing import JFrame
from javax.swing import JLabel
frame = JFrame(
 'HTMLtext_02',
 size = ( 200, 200 ),
 locationRelativeTo = None,
 defaultCloseOperation = JFrame.EXIT_ON_CLOSE
 )
text = '<html>'
text += '<ul compact>'
text += '<li><font color="red">Red</font></li>'
text += '<li><font color="green">Green</font></li>'
text += '<li><font color="blue">Blue</font></li>'
text += '</ul>'
label = frame.add( JLabel( text ) )
frame.setVisible( 1 )
raw_input()
```

Modifying the HTML Text

In addition to static components, your applications might have some dynamic components that might be improved by the users of the HTML. Do you remember the JToggleButton component discussed in Chapter 8? Figure 15-7 shows one way to use HTML to make the button state more obvious to the users.

Figure 15-7. *HTMLtext_03.py sample output*

Listing 15-12 shows part of the HTMLtext_03.py script that was used to produce

this output. If you're uncomfortable with the statement in lines 24-29, don't worry. It uses Jython's flexibility to determine which color and text values should be used by indexing an array using the result of calling the button.isSelected() method.

Listing 15-12. Using HTML in a JToggleButton from HTMLtext_03.py

```
6|class HTMLtext_03( java.lang.Runnable ) :
7| def run( self ) :
8| frame = JFrame(
9| 'HTMLtext_03',
10| size = ( 100, 100 ),
11| locationRelativeTo = None,
12| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
13| )
14| text = '<html><font color="red">Off</font>'
15| label = frame.add(
16| JToggleButton(
17| text,
18| itemStateChanged = self.toggle
19| )
20| )
21| frame.setVisible( 1 )
22| def toggle( self, event ) :
23| button = event.getItem()
24| button.setText(
25| '<html><font color="%s">%s</font>' % [
26| ( 'red' , 'Off' ),
27| ( 'green', 'On' )
28| ][ button.isSelected() ]
29| )
```

Listing 15-13 shows some additional ways that this might be done.[16] I'm quite certain that you can come up with some other alternatives yourself.

Listing 15-13. Alternative Ways of Setting the JToggleButton Text if

```
button.isSelected() :
text = '<html><font color="red">Off</font>' else :
text = '<html><font color="green">On</font>' button.setText( text )
... or even ...
```

```
if button.isSelected() :
color, value = 'red', 'Off'
else :
color, value = 'green', 'On'
text = '<html><font color="%s">%s</font>' % ( color, value )
button.setText( text )
```

Identifying the Sections

Now that you've learned about the components that you can use to display text that's been marked up in HTML, it's time to return to Jsoup to see what you can do to improve your Javadoc application.

Note in Figure 15-4 that the HTML for a Swing class can be quite extensive. The application that I'm considering, at least for this chapter, can display a subset of the information that's available on the user selected entry, not the complete details.

Let's work on this in steps, so that you can learn as you go along. To begin, let's see what it takes to build a tiny application that will display the header text for a specific Javadoc page. The getHeaders class that is used in all three of the getHeader scripts is shown in Listing 15-14. As you can see, it is a simple application that uses a SwingWorker descendent class to populate a scrollable text area with the HTML heading tags from the JFrame Javadoc URL.

Listing 15-14. The getHeaders Class[17]

```
62|class getHeaders1( java.lang.Runnable ) :
63| def run( self ) :
64| frame = JFrame(
65| 'JFrame headers',
66| size = ( 500, 250 ),
67| locationRelativeTo = None,
68| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
69| )
70| textArea = JTextArea()
71| frame.add( JScrollPane( textArea ) ) 72| url =
'.../docs/api/javax/swing/JFrame.html' 73| headerTask( url, textArea ).execute()
74| frame.setVisible( 1 )
```

[16] This is a kind of a tip of the hat to Perl's "Tim Toady" (TMTOWTDI) motto at http://en.wikipedia.org/wiki/There%27s_ more_than_one_way_to_do_it and the "Zen of Python," which can be seen at http://www.python.org/dev/peps/pep-0020/.

[17]The complete URL of the JFrame Javadoc is http://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html.

Listing 15-15 shows the headerTask that uses the Jsoup library to retrieve the specified HTML, process it, and display the result in the user-specified area. If you watch closely while executing any of these getHeader scripts, you might be able to see the connecting... and processing... messages that this class produces.

It is important to note that this class uses the Jsoup items to process the retrieved HTML. The ones new to this application are found in the getPlainText(...) method, as shown on lines 38-42.

Listing 15-15. The headerTask Class from the getHeader1.py Script Files

```
38|class headerTask( SwingWorker ) :
39| def __init__( self, url, result ) :
40| self.url = url # URL to be retrieved
41| self.result = result
42| SwingWorker.__init__( self )
43| def getPlainText( self, element ) :
44| visitor = FormattingVisitor( self.url )
45| walker = NodeTraversor( visitor )
46| walker.traverse( element )
47| return visitor.toString()
48| def doInBackground( self ) :
49| try :
50| self.result.setText( 'connecting...' )
51| doc = Jsoup.connect( self.url ).get()
52| self.result.setText( 'processing...' )
53| self.text = self.getPlainText( doc )
54| except :
55| Type, value = sys.exc_info()[ :2 ]
56| Type, value = str( Type ), str( value )
57| print '\nError:', Type
58| print 'value:', value
59| self.result.setText( 'Exception: %s' % value )
```

```
def done( self ) :
self.result.setText( self.text )
```

The FormattingVisitor(...) method is a descendant of the Jsoup NodeVisitor class, and its implementation is shown in Listing 15-16. This is the only class in the three getHeader#.py scripts that changes. In this class, the application appends the text of any HTML header tags that it finds to be displayed in the application text area. **Listing 15-16.** The FormattingVisitor() Method from getHeader1.py

```
class FormattingVisitor( NodeVisitor ) :
def __init__( self, url ) :
self.result = StringBuilder()
def append( self, text ) :
newline = [ '', '\n' ][ self.result.length() > 0 ]
self.result.append( newline + text )
def head( self, node, depth ) :
name = node.nodeName()
if name in [ 'h1', 'h2', 'h3', 'h4', 'h5', 'h6' ] :
if node.hasText() :
self.append( '%s: %s' % ( name, node.text() ) )
def tail( self, node, depth ) :
name = node.nodeName()
def toString( self ) :
return str( self.result )
```

The only method that changes in the three getHeader scripts is head(...). The one from getHeader2.py is shown in Listing 15-17. Note how this iteration is only interested in the H3 (Header 3) tags.
Listing 15-17. The head(...) Method from getHeader2.py

```
def head( self, node, depth ) :
name = node.nodeName()
if name == 'h3' :
if node.hasText() :
self.append( '%s: %s' % ( name, node.text() ) )
```

Listing 15-18 shows the last version of the head(...) method. This version is only interested in H3 tags whose text doesn’t contain the word “inherited.”

Listing 15-18. The head(...) Method from getHeader3.py

```
29| def head( self, node, depth ) :
30| name = node.nodeName()
31| if name == 'h3' and node.hasText() :
32| text = node.text()
33| if text.find( 'inherited' ) < 0 :
34| self.append( '%s: %s' % ( name, text ) )
```

Figure 15-8 shows the output of these three getHeader scripts. From this output, it shouldn’t be too much of a leap to figure out how to change the Javadoc script to display the most useful H3 text. Before you do that, though, take a moment to figure out where you want to go next.

JFrame headers
h2: Class JFrame
h3: Nested Class Summary
h3: Nested classes/interfaces inherited from class java.awt.Frame
h3: Nested classes/interfaces inherited from class java.awt.Window
h3: Nested classes/interfaces inherited from class java.awt.Container
h3: Nested classes/interfaces inherited from class java.awt.Component
h3: Field Summary
h3: Fields inherited from class java.awt.Frame
h3: Fields inherited from class java.awt.Component
h3: Fields inherited from interface javax.swing.WindowConstants
h3: Fields inherited from interface java.awt.image.ImageObserver
h3: Constructor Summary

JFrame headers
h3: Nested Class Summary
h3: Nested classes/interfaces inherited from class java.awt.Frame
h3: Nested classes/interfaces inherited from class java.awt.Window
h3: Nested classes/interfaces inherited from class java.awt.Container
h3: Nested classes/interfaces inherited from class java.awt.Component
h3: Field Summary
h3: Fields inherited from class java.awt.Frame
h3: Fields inherited from class java.awt.Component
h3: Fields inherited from interface javax.swing.WindowConstants
h3: Fields inherited from interface java.awt.image.ImageObserver
h3: Constructor Summary
h3: Method Summary

JFrame headers
h3: Nested Class Summary
h3: Field Summary
h3: Constructor Summary
h3: Method Summary
h3: Field Detail
h3: Constructor Detail
h3: Method Detail

Figure 15-8. *Sample output from the getHeader scripts*

Fixing the Great Divide

One of the problems that I have with the way the javadocInfo_03.py application looks is the fact that the divider doesn't provide enough space to see the longer class names. How can you fix this?

To begin with, it's a good idea to adjust the position to something more convenient. If you do that, though, you need to have the application tell you the new position, so that you can use it as the starting position for the revised application.

You need to add a PropertyChangeListener event handler to the JSplitPane and watch for the right kind of changes. This is all very easy to say, but how easy is it to do? Listing 15-19 shows just how easy this can be. In line 109, you see how to specify the PropertyChangeListener event handler method, and in lines 137-146, you see how the event handler determines and displays the divider-related changes.

Listing 15-19. PropertyChangeListener Changes in javadocInfo_04.py

```
| ...
104| frame.add(
105| JSplitPane(
106| JSplitPane.HORIZONTAL_SPLIT,
107| pane,
108| JScrollPane( self.area ),
109| propertyChange = self.propChange
110| )
111| )
| ...
137| def propChange( self, pce ) :
138| src = pce.getSource() # Should be our JSplitPane
139| prop = pce.getPropertyName()
140| if prop == JSplitPane.LAST_DIVIDER_LOCATION_PROPERTY :
141| curr = src.getDividerLocation()
142| last = pce.getNewValue()
143| prev = pce.getOldValue()
```

```
144| print '\nlast: %4d prev: %4d curr: %4d' % (
145| last, prev, curr
146| ),
```

Figure 15-9 shows the sample output from an execution of the script that determined a more appropriate divider value.

```
last:   -1  prev:    0  curr:  129
last:  129  prev:   -1  curr:  129
last:   -1  prev:  129  curr:  129
last:  129  prev:   -1  curr:  271
last:  271  prev:  129  curr:  271
last:  129  prev:  271  curr:  271
```

Figure 15-9. *SplitPane PropertyChangeListener output*

So what can you do with this newfound knowledge? Well, my initial attempt consisted of adding a dividerLocation keyword argument to the JSplitPane[18] constructor call. Unfortunately, this caused an exception indicating that this is a read-only value.

Looking at the Javadoc, you can see that there are multiple setDividerLocation(...) methods, which is likely to be the source of the problem. So you need to call one of these setter methods after JSplitPane has been instantiated. Listing 15-20 shows how this is done in the javadocInfo_05.py script.

Listing 15-20. Using a DividerLocation Setter Method Call

```
106| sPane = JSplitPane(
107| JSplitPane.HORIZONTAL_SPLIT,
108| pane,
109| JScrollPane( self.area ),
110| propertyChange = self.propChange
111| )
112| sPane.setDividerLocation( 234 )
```

Filtering the List

One of the problems with the complete list of classes that's initially displayed by the javadocInfo script is that it's too long. If you drag the scroll bar down, you'll quickly see that there are something like 4,000 classes listed. This makes it harder for the users to deal with.

Wouldn't it be nice to use the JTextField as an input field filter that allowed users to enter text to be matched? That's what you'll do next. The changes that need to be made include the following:

• Locate the places in the script where the setText(...) method is used to show the URL associated with the user selection.

• Add a listener to JTextField that allows you to monitor user input and locate class names containing the user-specified text. For this, I chose to use a CaretListener and check for changes in the user-specified text.

The javadocInfo_06.py script contains these changes and Figure 15-10 shows some sample screens. [18]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JSplitPane.html.

Figure 15-10. *javadocInfo_06.py sample output with list filtering*

javadocInfo
JFileChooser
JFormattedTextField
JFormattedTextField.AbstractFormatter
JFormattedTextField.AbstractFormatterF
JFrame
JF
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<!-- NewPage -->
<html lang="en">
<head>
<!-- Generated by javadoc (version 1.6.0_18) on Wed Aug 01 12:25:52 PDT 2012 -->
<title>JFrame (Java Platform SE 7)</title>
<meta name="date" content="2012-08-01" />
<meta name="keywords" content="javax.swing.JFrame class" />
<meta name="keywords" content="EXIT_ON_CLOSE" />
<meta name="keywords" content="rootPane" />
<meta name="keywords" content="rootPaneCheckingEnabled" />
<meta name="keywords" content="accessibleContext" />
<meta name="keywords" content="frameInit()" />
<meta name="keywords" content="createRootPane()" />
<meta name="keywords" content="processWindowEvent()" />
<meta name="keywords" content="setDefaultCloseOperation()" />
<meta name="keywords" content="getDefaultCloseOperation()" />
<meta name="keywords" content="setTransferHandler()" />
<meta name="keywords" content="getTransferHandler()" />
<meta name="keywords" content="update()" />
<meta name="keywords" content="setJMenuBar()" />
<meta name="keywords" content="getJMenuBar()" />
<meta name="keywords" content="isRootPaneCheckingEnabled()" />
<meta name="keywords" content="setRootPaneCheckingEnabled()" />
<meta name="keywords" content="addImpl()" />
<meta name="keywords" content="remove()" />
<meta name="keywords" content="setLayout()" />
<meta name="keywords" content="getRootPane()" />
<meta name="keywords" content="setRootPane()" />
<meta name="keywords" content="setIconImage()" />
<meta name="keywords" content="getIconImage()" />
<meta name="keywords" content="setJMenuBar(JMenuBar)" />
<meta name="keywords" content="getContentPane()" />
<meta name="keywords" content="setContentPane()" />
<meta name="keywords" content="isFocusTraversalPolicyProvider()" />
<meta name="keywords" content="setFocusTraversalPolicyProvider()" />
<meta name="keywords" content="getGlassPane()" />
<meta name="keywords" content="setGlassPane()" />
<meta name="keywords" content="getLayeredPane()" />
<meta name="keywords" content="setLayeredPane()" />
<meta name="keywords" content="getJMenuBar()" />
<meta name="keywords" content="setJMenuBar()

JFrame headers

Nested Class Summary | Field Summary | Constructor Summary | Method Summary

Nested Classes

Modifier and Type	Class and Description
protected class	JFrame.AccessibleJFrame This class implements accessibility support for the JFrame class.

JFrame headers

Nested Class Summary | Field Summary | Constructor Summary | Method Summary

Fields

Modifier and Type	Field and Description
protected AccessibleContext	accessibleContext The accessible context property.
static int	EXIT_ON_CLOSE The exit application default window close operation.
protected JRootPane	rootPane The JRootPane instance that manages the contentPane and

Figure 15-11. *getHeaders5.py output with JFrame h3 tables on JTabbedPane*

The most significant changes are shown in Listing 15-21, which contains the FormattingVisitor class from the getHeaders5.py script.

Listing 15-21. FormattingVisitor Class Used to Build the JTabbedPane

```
25|class FormattingVisitor( NodeVisitor ) :
26| def __init__( self, url ) :
27| self.Tabs = JTabbedPane()
28| self.header = re.compile( '[hH][1-6]$' )
29| self.h3Text = ''
30| def head( self, node, depth ) :
31| name = node.nodeName()
32| if re.match( self.header, name ) :
33| self.h3Text = [ '', node.text() ] [ name[ 1 ] == '3' ]
34| def tail( self, node, depth ) :
```

```
35| name = node.nodeName()
36| if self.h3Text and name == 'table' :
37| ePane = JEditorPane(
38| 'text/html', # mime type
39| '<html>' + str( node ), # content
40| editable = 0
41| )
42| self.Tabs.addTab( self.h3Text, JScrollPane( ePane ) )
43| def toString( self ) :
44| tp = self.Tabs
45| return '\n'.join(
46| [
47| tp.getTitleAt( i )
48| for i in range( tp.getTabCount() )
49| ]
50| )
```

Table 15-2 explains how this class works, in detail.

Table 15-2. *FormattingVisitor Class, Explained* **Lines Description**

26-29 Class constructor used to instantiate the JTabbedPane, as well as define a regular expression (RegExp) to identify heading tags, specifically h3 tags.

30-33 The head(...) method is called by the NodeTraversor() method as the HTML is processed. It is invoked when any open tag is encountered (such as <h2> or <table...>). The value of the h3Text variable is set to an empty string (when any other head level is encountered) or to the text associated with the heading tag when an h3 tag is found.

34-42 The tail(...) method is called as the HTML is processed, specifically when an end tag (such as </h3> or </table>) is found. When a table “end” tag is found after an <h3> tag, a read-only JEditorPane is created containing the whole <table> HTML. This JEditorPane instance is placed in a JScrollPane and added as a tab to the JTabbedPane.

43-50 The toString(...) method is provided and returns a string with all of the JTabbedPane tab names, separated by newline \n characters.

The only other significant difference in this script relates to how the

SwingWorker descendent class (headerTask) updates the application in a way that allows the updates to be shown to the users. Listing 15-22 shows the headerTask class from getHeader5.py.

Listing 15-22. The headerTask Class from the getHeaders5.py Script

```
51|class headerTask( SwingWorker ) :
52| def __init__( self, url, frame ) :
53| self.url = url # URL to be retrieved
54| self.frame = frame
55| SwingWorker.__init__( self )
56| def doInBackground( self ) :
57| try :
58| print 'connecting...'
59| doc = Jsoup.connect( self.url ).get()
60| print 'processing...'
61| visitor = FormattingVisitor( self.url )
62| walker = NodeTraversor( visitor )
63| walker.traverse( doc )
64| self.frame.add( visitor.Tabs )
65| self.frame.validate()
66| except :
67| Type, value = sys.exc_info()[ :2 ]
68| Type, value = str( Type ), str( value )
69| print '\nError:', Type
70| print 'value:', value
71| def done( self ) :
72| print 'done'
```

Table 15-3 describes the headerTask class in detail. The most important part of this is how this class informs the Swing framework that it needs to validate the application (frame) updates. If this isn't done, the user won't see any of the changes.

Table 15-3. *The headerTask Class, Explained*

Lines	Description
52-55	Class constructor used to save the specified URL, as well as a reference to the application frame to be modified. This method also has to invoke the

SwingWorker constructor in order to properly initialize the class as a SwingWorker instance.

56-70 The doInBackground() method is where the actual (generally long-running) processing occurs.
59 This is where the HTML from the specified URL is retrieved.
61-63 A NodeVisitor call uses the FormattingVisitor instance to process the Javadoc from the specified URL.

64 The result of the processing is a JTabbedPane instance, which is added to the application's JFrame instance. Unfortunately, this call to the frame.add(...) method is not likely to happen in a way that lets the Swing framework know that a change has occurred.

65 The call to frame.validate() forces the framework to recognize that it needs to process the changes and update the user display.

Note: Lines 58, 60, and 72 write messages to the console to let you know that things are progressing. They certainly aren't required and may be commented out.
If you comment out line 72, you also need to add a pass statement to satisfy language syntax requirements.

The Jsoup library is quite powerful. Even a tiny bit of testing shows some neat capabilities. For example, I find the output shown in Figure 15-11 a bit hard to read. When I looked at the HTML found in the Javadoc pages, I saw that all of the tables have a border="0" attribute that keeps the grid lines between columns and rows from being shown. Do you know how difficult it is to use the Jsoup library to change the table border attributes to "1"?

Take a look at Listing 15-22, specifically at line 54 where the HTML document is retrieved from the remote website. All you need to do is add the statement in Listing 15-23 after the HTML has been retrieved, and before it is processed by the rest of the code.

Listing 15-23. Changing the border Attribute of All the Tables doc.select('table').attr('border', '1')
It's as simple as that. The result of adding this statement can be seen in Figure 15-12.

Figure 15-12. *getHeaders6.py output with JFrame h3 tables on JTabbedPane*

Figure 15-12. (*continued*)

If you think about it, using the statement in Listing 15-23 to find and change all of the table border attributes might be extra work. This would be true if there were other tables in the document that aren't associated with the h3 sections that you want to display. The alternative is to change the attribute only on those tables that you will be adding to the JEditorPane in the tail(...) method of the FormattingVisitor class (shown in Listing 15-21). If you do this, you need to add the statement in Listing 15-24 between lines 31 and 32 of Listing 15-21.

Listing 15-24. Changing the border Attribute of the Current Table

```
node.attr( 'border', '1' )
```

Adding Tabbed Editor Panes to the Javadoc Application

What do you need to do in order to add the functionality of the getHeader6.py script to the latest javadocInfo_06.py script?

• The right pane needs to be changed. It isn't a simple JTextArea instance any longer. You have to decide what you want to display when nothing is selected, and you have to figure out how to display the JTabbedPane after a class' HTML has been processed.

• What do you want the application to do as the data is being retrieved and processed? Currently, you can display a status message in the JTextArea. It would seem that you need to dynamically change the right panel of SplitPane, based on what needs to be displayed.

In order for the application to display different kinds of panes in the frame, you need to use a JSplitPane method that wasn't covered in Chapter 5. Since the SplitPane has left and right parts, you need to use the setRightComponent(...) method to dynamically replace that part of the application.

In order for the application to display tabbed JEditorPanes, you'll also need to replace the textTask class (as seen in Listing 15-9) with a modified version of the headerTask class shown in Listing 15-22. This class needs to have the right information so that it can display a status message or the final JTabbedPane result in the application's SplitPane.

What's the result of making these changes? Figure 15-13 shows some sample images from the modified script (javadocInfo_07.py).

Figure 15-13. *javadocInfo_07.py sample output*

Figure 15-13. (*continued*)

Were there any surprises that happened during this combination process? A few, but not nearly as many as I feared. The biggest surprise was that the first attempt showed that when the JTabbedPane replaced the right component of the SplitPane, the width of the left component used to display the list of classes

narrowed drastically. So, I had to set the minimum size of the JScrollPane containing this list.

What Improvements/Enhancements Remain?

As is frequently the case when using any program, possible enhancements come to mind. You might want to consider adding some menu items. This would make sense if you wanted to allow the users to specify the URL of the page from which it should load the list of Java classes. Another possible menu entry might be used to allow the user to filter class entries (such as AWT and Swing).

You might also want to add a tab to the tabbed pane that shows the class hierarchy produced by the classInfo function mentioned in Chapter 4.
If you are really ambitious, you could add functionality to the Editor Panes so that the links in the displayed HTML open the specified website in a browser.
I haven't delved too deeply into the subject of Editor Panes. There is a decent page in the Java Swing Tutorials entitled "How to Use Editor Panes and Text Panes"[19] that you might want to take a look at.

Summary

One of the useful things about this chapter is that it demonstrates how easy it is to use existing Java classes and libraries. There is an enormous number of class libraries on the Internet. Hopefully, this chapter gives you the courage to use them in your Jython scripts.

The next chapter discusses dialog boxes and how they can be used to interact with users to obtain input for scripts.

[19]See http://docs.oracle.com/javase/tutorial/uiswing/components/editorpane.html.

Chapter 16

Conversing with a User with Dialog Boxes

According to several dictionaries, a "dialogue" is an exchange of information between two entities, at least one of whom is a person. Maybe that's why a window for providing input to an application is called a "dialog box."

Way back in Chapter 1, you read all about the top-level containers. The examples throughout the book have used the JFrame class as the top-level container of choice. Now you're going take some time to learn about the JDialog[1] container class and dialog boxes in general. This chapter also touches on a few related issues. In Chapter 17, you'll continue by learning about some specialized dialog boxes that you can use to make your life as an application developer much easier.

This chapter is all about using dialog boxes to get information from the user into an application. It starts by discussing where dialog boxes fit in the Swing class hierarchy. Then is discusses how you position them, especially when multiple displays exist. Additionally, you'll see how various dialog boxes can be created using JOptionPane methods.

What Are Dialog Boxes?

Dialog boxes are separate windows that are used to convey information to, and from, the application user. They can also be used to interact with the user to obtain input that is returned to the application. You are likely very familiar with the kind of message windows that display informational, warning, and error messages. It is also likely that you've used a dialog box to provide information to applications. Now you'll see what you need to do to use these dialog boxes in your applications.

What's a JDialog?

Listing 16-1 shows the class hierarchy for the JDialog class. It's interesting to note how similar this hierarchy is to the JFrame class,[2] shown in Listing 16-2.
[1]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JDialog.html. [2]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html. **Listing 16-1.** JDialog Class Hierarchy

```
wsadmin>from javax.swing import JDialog wsadmin>
wsadmin>classInfo( JDialog )
javax.swing.JDialog
| java.awt.Dialog
| | java.awt.Window
| | | java.awt.Container
```

```
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver | | | | | java.awt.MenuContainer | | | | |
java.io.Serializable
| | | javax.accessibility.Accessible | javax.swing.WindowConstants
| javax.accessibility.Accessible | javax.swing.RootPaneContainer
wsadmin>
```

This similarity should help you understand that JDialog instances should be able to contain the same kind of Swing components that JFrame instances do.

Listing 16-2. JFrame Class Hierarchy

```
wsadmin>from javax.swing import JFrame wsadmin>
wsadmin>classInfo( JFrame )
javax.swing.JFrame
| java.awt.Frame
| | java.awt.Window
| | | java.awt.Container
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver | | | | | java.awt.MenuContainer | | | | |
java.io.Serializable
| | | javax.accessibility.Accessible | | java.awt.MenuContainer
| javax.swing.WindowConstants
| javax.accessibility.Accessible | javax.swing.RootPaneContainer wsadmin>
```

These similarities made me wonder just how similar the two top-level containers might be as well. Using the final version of the javadocInfo.py script file discussed in Chapter 15,[3] take a quick look at the JDialog methods shown in Figure 16-1.

[3]Which is ...\Code\Chap_15\javadocInfo_07.py.

Figure 16-1. *JDialog methods*

It also makes me feel good about the effort that was invested in creating that script/application. After starting it, you only need to type two letters and use two mouse clicks to display the information about the class methods:

1. Type JD to filter the list of classes.
2. Select the JDialog entry on the short list of classes.
3. Select the Method Summary tab. You’re done!

A quick glance at these methods shows that the JFrame and JDialog classes both have RootPanes, ContentPanes, GlassPanes, MenuBars, and many other things in common. There are a number of differences as well. If you start by looking at the class constructors for each class, it should quickly become obvious that there are a lot more JDialog constructors than there are JFrame constructors. It seems that this is the result of the presence in the JDialog constructors of two additional parameters:

• The owner argument (dialog, frame, or window)
• The modality argument (a Boolean flag or a Dialog.ModalityType)

What’s the GraphicsConfiguration Component Do?

One issue that pops up time and time again in the Javadocs for various Swing-

related classes is that they contain so much information that it is often difficult to comprehend. This is a long way of saying that the first few (perhaps dozen) times that I looked at the JFrame and JDialog Javadoc pages, I didn't take the time or effort to see the GraphicsConfiguration[4] parameter in some of the constructors.

[4]See http://docs.oracle.com/javase/8/docs/api/java/awt/GraphicsConfiguration.html.

When I finally did see it, it took me a while to realize how useful this particular parameter can be. However, this is only the case when the users can have multiple displays. You can use this parameter to identify on which of the available displays the application will open a JFrame or JDialog instance.

Listing 16-3 shows one technique for determining the position, width, and height of the available displays using the GraphicsEnvironment class. Did you notice that this class comes from java.awt and not the javax.swing library?

Listing 16-3. Using the GraphicsEnvironment Class

```
wsadmin>from java.awt import GraphicsEnvironment
wsadmin>
wsadmin>LGE = GraphicsEnvironment.getLocalGraphicsEnvironment()
wsadmin>for SD in LGE.getScreenDevices() :
wsadmin> for GC in SD.getConfigurations() :
wsadmin> print GC.getBounds()
wsadmin>
java.awt.Rectangle[x=1920,y=0,width=1920,height=1080]
java.awt.Rectangle[x=0,y=0,width=1920,height=1080]
wsadmin>
```

Table 16-1 describes each of the statements in the interactive wsadmin session shown in Figure 16-1.

Table 16-1. *Using the GraphicsEnvironment Session*

Lines Description

1 Add the GraphicsEnvironment class from the java.awt library to the local namespace.

3 Use the getLocalGraphicsEnvironment() method to obtain information about

the local graphics environment.

4 Loop over the ScreenDevice objects in the environment.
5 Loop over the GraphicConfiguration objects in each ScreenDevice object.
6 Display the bound instance values for the current GraphicConfiguration object.
8 - 9 Results generated by the print statement in line 6.

The output shown in Listing 16-3 is specific to my environment and shows that it has two displays, logically configured to be side by side, with the primary display on the right. How do you tell this? Well, the primary display (“Screen 0”) is considered to be at location [x=0, y=0]. The next display (“Screen 1”) has a positive x value, which indicates that it is logically to the right of the primary display. If a third display were present that was logically considered to be on the left of the primary display, it would have a negative value of x. Adding the values of x and width from the second display (1920 + 1920 = 3840) results in the logical width of the combined display.

Using a GraphicsConfiguration Object

Listing 16-4 shows how to obtain a GraphicsConfiguration object from a ScreenDevice object. Once you have it, how can it be used? The JDialog and JFrame classes both have constructors that include GraphicsConfiguration objects. So, once you have a configuration object that identifies a display, this can be provided on the JDialog or JFrame constructor call to identify the display with which the application should be associated. Listing 16-4 shows an example class that uses a GraphicsConfiguration object to identify the user display on which the JFrame instance should be displayed.

Listing 16-4. Using a GraphicsConfiguration Object

8|class ScreenLoc(java.lang.Runnable) :
9| def run(self) :
10| d = 0
11| LGE = GraphicsEnvironment.getLocalGraphicsEnvironment() 12| for GD in LGE.getScreenDevices() :
13| CO = GD.getDefaultConfiguration()
14| for GC in GD.getConfigurations() :
15| b = GC.getBounds()
16| frame = JFrame(

```
17| 'Screen: %d' % d,
18| CO,
19| size = (
20| int( b.getWidth() ) >> 1,
21| int( b.getHeight() ) >> 3
22| ),
23| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 24| )
25| d += 1
26| frame.add( JLabel('CO' ) )
27| frame.setVisible( 1 )
```

Table 16-2 explains the ScreenLoc class shown in Listing 16-4 in detail. One way to figure out what the code is doing is to keep an eye out while viewing the Javadoc pages of the code snippets. In this case, some of this is explained on the GraphicsConfiguration and GraphicsDevice pages, which help explain how they can be used. The ScreenLoc class in Figure 16-1 is based on these code snippets.

Table 16-2. *ScreenLoc Class Explained*
Lines Description

8-9 Define the class as a descendent of the java.lang.Runnable class; its run() method will be called by the Swing Dispatch thread.
10 d is a simple integer identifying the device number 0..N-1.
11 Call the static getLocalGraphicsEnvironment() method from the GraphicsEnvironment[5] class to obtain information about the local graphics environment (LGE).

12-26 Process each GraphicsDevice[6] (GD) in the local graphics environment.
13 Get the default GraphicsConfiguration object (CO) associated with the current device.
14-24 Process all of the GraphicsConfiguration (GC) objects for the current GraphicsDevice.
15 Get the bounds (java.awt.Rectangle[7]) describing the device coordinates.
16-23 Create a JFrame instance on the current GraphicsDevice using the configuration object.

Note: Since only the size is provided, the JFrame instance will be positioned in the top-left corner of the display.

19-22 The size of the frame is defined to be half the display width and an eighth of the display height.
24 Increment the display counter (d).
25 Add a JLabel instance to the current frame containing the information about the configuration object.
26 Make the frame instance visible.

Is a GraphicsConfiguration Object Really Necessary?

If you want to create a frame or a dialog window using a configuration object, you can. But it would be just as easy to use the device bounds rectangle to position the window, especially in Jython. Listing 16-5 shows the ScreenPos class, from the script of the same name, which shows how easily this can be done.

Listing 16-5. Using Device Bounds to Position Items

```
8|class ScreenPos( java.lang.Runnable ) :
9| def run( self ) :
10| d = 0
11| LGE = GraphicsEnvironment.getLocalGraphicsEnvironment() 12| for GD in
LGE.getScreenDevices() :
13| for GC in GD.getConfigurations() :
14| b = GC.getBounds()
15| w = int( b.getWidth() ) >> 1
16| h = int( b.getHeight() ) >> 3
17| x = int(
18| ( int( b.getWidth() - w ) >> 1 ) + b.getX() 19| )
20| y = int(
21| ( int( b.getHeight() - h ) >> 1 ) + b.getY() 22| )
23| frame = JFrame(
24| 'Screen: %d' % d,
25| bounds = ( x, y, w, h ),
26| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 27| )
28| d += 1
29| frame.add( JLabel('GC' ) )
30| frame.setVisible( 1 )
```

[5] See

http://docs.oracle.com/javase/8/docs/api/java/awt/GraphicsEnvironment.html.
[6]See http://docs.oracle.com/javase/8/docs/api/java/awt/GraphicsDevice.html.
[7]See http://docs.oracle.com/javase/8/docs/api/java/awt/Rectangle.html.

Table 16-3 explains the statements shown in Listing 16-5 in detail. Remember that these listings sometimes have more lines than might normally be used only because of the narrow width required for these pages.

Table 16-3. *ScreenPos Example, Explained*
Lines Description

8-9 Define the class as a descendent of the java.lang.Runnable class; its run() method will be called by the Swing Dispatch thread.
10 d is a simple integer identifying the device number 0..N-1.
11 Call the static getLocalGraphicsEnvironment() method from the GraphicsEnvironment class to obtain information about the local graphics environment (LGE).

12-29 Process each GraphicsDevice (GD) in the local graphic environment.
13-29 Process all of the GraphicsConfiguration (GC) objects for the current GraphicsDevice.
14 Get the bounds (b) Rectangle defining the GraphicsDevice.

15 The width (w) of the window to be instantiated should be half the width of the graphic device on which it will be seen.
16 The height (h) of the window should be an eighth the height of the display screen.
17-19 The window's upper-left corner (the X coordinate) is half the screen width minus the width of the window. Adding the screen's X coordinate positions it properly.
20-22 The window's upper-left corner (the Y coordinate) is half the screen height minus the height of the window. Adding the screen's Y coordinate positions it properly.

23-26 Instantiate the JFrame using the given title and the previously computed coordinates.
27 Increment the display counter (d).

28 Add a JLabel instance to the current frame containing the information about

the GraphicsConfiguration object.
29 Make the frame instance visible.

The result of all of this simply shows that using a GraphicsConfiguration object isn't required. It is generally easier to use the Rectangle object available with the GraphicsDevice to determine where to position the window, based on the window and screen sizes.

What About an Owner?

In the section entitled, "What's a JDialog?," I mentioned that there are two parameters that exist in most of the JDialog constructors that don't exist in the JFrame constructors. These are the owner and modality parameters, and they are closely associated. When a JDialog instance is configured to be modal and the window is visible, all user input is blocked to all of the other application windows. This forces the user to interact with the modal dialog window first.

■**Note** if the owner parameter is specified as None, then a shared, hidden frame will be identified as the owner.
Figure 16-2 shows some sample output generated by the SimpleDialog.py script. It shows how you can create multiple non-modal dialog windows, but only one modal one.

Figure 16-2. *SimpleDialog sample output*

Listing 16-6 shows the SimpleDialog class from this application. It only needs an ActionListener event handler to respond to when one of the application buttons is clicked. Note how the JDialog constructor specifies the owner using a value of None (line 36). This forces the Swing library to use a (shared) hidden frame, which forces the modal dialog box to be handled first.

Listing 16-6. The SimpleDialog Class from SimpleDialog.py

```
9|class SimpleDialog( java.lang.Runnable ) :
10| def run( self ) :
11| self.frame = frame = JFrame(
```

```
12| 'SimpleDialog',
13| size = ( 250, 100 ),
14| layout = GridLayout( 0, 2 ),
15| locationRelativeTo = None,
16| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
17| )
18| frame.add(
19| JButton(
20| 'Modal',
21| actionPerformed = self.makeDialog
22| )
23| )
24| frame.add(
25| JButton(
26| 'non-Modal',
27| actionPerformed = self.makeDialog
28| )
29| )
30| self.boxNum = 0
31| frame.setVisible( 1 )
32| def makeDialog( self, event ) :
33| cmd = event.getActionCommand()
34| isModal = ( cmd == 'Modal' )
35| dialog = JDialog(
36| self.frame, # Try None as owner... 37| title = '%d: %s' % ( self.boxNum, cmd
), 38| modal = isModal,
39| size = ( 200, 100 ),
40| locationRelativeTo = self.frame
41| )
42| self.boxNum += 1
43| dialog.add( JLabel( cmd ) )
44| dialog.setVisible( 1 )
```

What happens if you iconify the application when non-modal dialog windows exist? Well, if they were instantiated with the owner parameters specified as None, nothing happens to them. They remain as they are. However, if the application is identified as the owner (in this case specifying an argument of self.frame), then all of the associated non-modal windows will be iconified as well. And if you deiconify the application, all of the non-modal windows will

follow suit. Knowing this might help you decide if you want to specify the owner argument when you instantiate your JDialog instance.

Where's the Dialog?

At the start of this chapter, you saw that a dialog is about communication. Up to now, you haven't done anything with the JDialog instances that you've created other than make them appear and disappear. Where's the communication in that? This section explains how you enable a JDialog instance to communicate with the application or the component that created it.

Let's start with the simple case. If your application creates a modal JDialog instance, the application will be in suspended animation until the dialog is hidden or closed. Since the application created the object instance, it can now call any of its getter methods to retrieve information from the object. Figure 16-3 shows some sample output from a trivial application that uses a custom JDialog box to obtain user input.

Figure 16-3. *CustomDialog1.py sample output*

Listing 16-7 shows the CustomDialog class from this sample application. Since there are some points of interest, this section will describe in more detail how this application works.

Listing 16-7. CustomDialog Sample Class

```
12|class CustomDialog( java.lang.Runnable ) :
13| def run( self ) :
14| self.frame = frame = JFrame(
15| 'CustomDialog',
```

```
16| size = ( 200, 100 ),
17| locationRelativeTo = None,
18| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
19| )
20| self.label = frame.add( JLabel( '' ) )
21| frame.add(
22| JButton(
23| 'Prompt user',
24| actionPerformed = self.popup
25| ),
26| BorderLayout.SOUTH
27| )
28| frame.setVisible( 1 )
29| def popup( self, event ) :
30| self.dialog = dialog = JDialog(
31| self.frame,
32| 'Name prompt',
33| 1,
34| layout = GridLayout( 0, 2 ),
35| locationRelativeTo = self.frame
36| )
37| dialog.add( JLabel( "What's your name?" ) )
38| self.text = dialog.add(
39| JTextField(
40| 10,
41| actionPerformed = self.enter
42| )
43| )
44| self.result = None
45| dialog.pack()
46| dialog.setVisible( 1 )
47| self.label.setText( 'Name = "%s"' % self.result )
48| def enter( self, event ) :
49| self.result = self.text.getText()
50| self.dialog.setVisible( 0 )
```

Table 16-4 describes the CustomDialog class shown in Listing 16-7 in more detail. For simplicity's sake, the code that's similar to other examples is described in less detail.

Table 16-4. *CustomDialog Class, Explained*

Lines	Description
13-28	CustomDialog run() method that populates the JFrame instance and displays it for the user.
20	The popup() ActionListener method needs to access the label field, so addressability to it must be saved.
29-47	popup() ActionListener event handler, which is invoked when the button is activated. 30-36 Create a modal JDialog instance that identifies the application frame as the owner and tell the Swing library to display the JDialog instance relative to it.
37-46	Populate the JDialog and make it visible.
47	This is the point where control returns when the JDialog is no longer visible (it's either closed or hidden).
48-50	ActionListener event handler for the JDialog user input field (JTextField). **Note:** This event handler expects to be associated with a modal dialog instance. It does not include any protection from concurrency issues.

The last row of Table 16-4 includes one of the significant reasons for using modal dialog boxes. Dealing with concurrency issues can add complexity to your applications. So think carefully before using non-modal dialog boxes that can update values in the component that creates it.

Multiple Modal Dialog Boxes Are Annoying

Please be considerate of your users. Even though it's possible for your application to create multiple modal dialog boxes, don't do it. Unfortunately, I see this quite often, especially when a web application uses Adobe Flash. I tend to see something like the image shown in Figure 16-4. Unfortunately, these boxes can stack up, so I find myself clicking on the Deny ("No") button a lot. This can be frustrating. As a user, I try to avoid using this kind of application whenever possible. Keep this in mind when you create your applications. Don't create a "user-hostile" application that does something annoying like this.

Figure 16-4. *Please take "no" for an answer*

Using JOptionPane Methods

It's commonplace to come across applications that use modal dialog boxes to obtain user input, so it should be no surprise that there's an easy way to create a variety of commonly used dialog boxes.

There are four major and two minor variants of methods in the JOption class for creating modal dialog boxes to obtain user input. All of these methods have names that adhere to the following naming convention: show%1%2Dialog(), where %2 represents the major portion, which will be one of the following four types:

• Input
• Confirm
• Message
• Option

And %1 represents the minor portion, which will be empty or internal. This means that you should be able to produce the complete list of method names by using a simple nested loop, right? Listing 16-8 shows this simple list.
Listing 16-8. JOption show*Dialog Method Name Variants

```
wsadmin>num = 1
wsadmin>for minor in [ '', 'Internal' ] :
wsadmin> for major in [ 'Input', 'Confirm', 'Message', 'Option' ] : wsadmin> print
'%2d: show%s%sDialog()' % ( num, minor, major ) wsadmin> num += 1
wsadmin>

1: showInputDialog()
2: showConfirmDialog()
3: showMessageDialog()
```

4: showOptionDialog()
5: showInternalInputDialog()
6: showInternalConfirmDialog()
7: showInternalMessageDialog()
8: showInternalOptionDialog()
wsadmin>

Unfortunately, it doesn't take into consideration the fact that Java allows method overloading, so each method name can have multiple signatures. What does this mean for Jython programmers? It means that you have to take this into consideration.

■**Note** For now, ignore the showInternal variants. You'll see them again in Chapter 19, where you learn about JInternalFrames.

The JOptionPane.showMessageDialog() Method

The simplest dialog box that can be displayed by the JOptionPane class is called a MessageDialog. It is the simplest primarily because it returns no value. The purpose of a MessageDialog is to inform users about some event. Since it doesn't provide the user with any opportunity to provide input, the showMessageDialog() method doesn't have to return any value.

Figure 16-5 shows the Java method signature and a simple image from the MessageDialogDemo.py application.

Notice that the dialog box has a simple title of "Message," an informational icon, and an OK button.

```
showMessageDialog(
    Component parentComponent,
    Object    message
)
```

Figure 16-5. *Simple MessageDialog signature and sample image*

Figure 16-6 shows another showMessageDialog() method signature, as well as an example image from the MessageDialogDemo.py application. This version of the method allows you to specify the title to be displayed in the title bar. It also allows you to specify the messageType and an optional icon to be displayed. If an icon isn't provided, then the icon that is displayed on the MessageDialog is based on the messageType that was specified.

```
showMessageDialog(
    Component parentComponent,
    Object    message,
    String    title,
    int       messageType,
    Icon      icon
)
```

Figure 16-6. *Complete MessageDialog signature and sample image*

Figure 16-7 shows the list of MessageType constants that are defined in the JOptionPane class. These should be used for messageType arguments in the show*Dialog() method calls. If you don't provide an icon parameter, the icon displayed on the dialog box will be based on the MessageType.

```
PLAIN_MESSAGE

ERROR_MESSAGE

INFORMATION_MESSAGE

WARNING_MESSAGE

QUESTION_MESSAGE
```

Figure 16-7. *MessageType constants defined in the JOptionPane class*

The JOptionPane.showOptionDialog() Method

The next most useful show*Dialog method is showOptionDialog(). It is useful, and frequently used, because of its flexibility. It can display a custom dialog. Figure 16-8 shows the list of parameters that should be specified when this method is called.

```
showOptionDialog(
    Component parentComponent,
    Object    message,
    String    title,
    int       optionType,
    int       messageType,
    Icon      icon,
    Object[]  options,
    Object    initialValue
)
```

Figure 16-8. *Complete showOptionDialog signature*

This method creates a dialog with a specified icon, where the initial choice is

determined by the initialValue parameter and the number of choices is determined by the optionType parameter.
You saw how the messageType value can be used to determine the icon to be shown if it's not specified. How is the optionType parameter used and what values can be specified? Figure 16-9 shows the JOptionPane constants that can be used for the optionType argument.

```
DEFAULT_OPTION

YES_NO_OPTION

YES_NO_CANCEL_OPTION

OK_CANCEL_OPTION
```

Figure 16-9. *OptionType constants defined in the JOptionPane class*

It would seem that you can use the optionType and messageType arguments to tell the showOptionDialog() method how many buttons to display, as well as their labels. Listing 16-9 shows the simplest form of a call to the showOptionDialog() method. Note how you can use an optionType of DEFAULT_OPTION and None for the options and initialValue arguments.

Listing 16-9. Sample showOD() Method Used to Call showOptionDialog

```
26| def showOD( self, event ) :
27| options = 'Bacon,Eggs,Spam'.split( ',' )
28| result = JOptionPane.showOptionDialog(
29| self.frame, # parentComponent
30| 'What goes good with spam?', # message text
31| 'This is a test!', # title
32| JOptionPane.DEFAULT_OPTION, # optionType 33|
JOptionPane.QUESTION_MESSAGE, # messageType 34| None, # icon 35|
None, # options 36| None # initialValue 37| )
38| self.label.setText( 'result = %d' % result )
```

Figure 16-10 shows the different buttons that are displayed on the OptionDialog when you specify the various constants listed in Figure 16-9.

Figure 16-10. *Simple OptionDialog examples*

What if you don't like these kinds of buttons? Can you specify button labels other than Yes, No, OK, and Cancel? Sure, that's what the options and initialValue arguments are for. Listing 16-10 shows how to provide your own button labels for the OptionDialog box.

Listing 16-10. Modified showOD() Method Specifying Options

```
26| def showOD( self, event ) :
27| options = 'Bacon,Eggs,Spam'.split( ',' )
28| result = JOptionPane.showOptionDialog(
29| self.frame, # parentComponent
30| 'What goes good with spam?', # message text
31| 'This is a test!', # title
32| JOptionPane.DEFAULT_OPTION, # optionType
33| JOptionPane.QUESTION_MESSAGE, # messageType
34| None, # icon
35| options, # options
36| options[ -1 ] # initialValue
37| )
38| self.label.setText( 'result = %d' % result )
```

Figure 16-11 shows the OptionDialog that results from this showOD() method.

Figure 16-11. *OptionDialog example using non-standard button labels*

Are you limited to one, two, or three buttons? As a matter of fact, you're not. If you provide an array (list) of options with more than three values, the showOptionDialog() is able to show them to you.
The example OptionDialog shown in Figure 16-12 can easily be produced by changing line 27 in Listing 16-10 to the statement in Listing 16-11.

Figure 16-12. *OptionDialog example with eight buttons*

Listing 16-11. Specifying Many OptionDialog Button Labels

```
27| options = 'Now is the time for all good spam'.split( ' ' )
```

Certainly this example is a bit excessive, but it does demonstrate an interesting point. It should also raise a question or two. When you display a variety of buttons on a dialog box, how can you tell which one the user pressed? If you use one of the standard OptionDialog objects, you can use the return value constants defined in the JOptionPane class and listed in Figure 16-13 to make this determination.

```
YES_OPTION

NO_OPTION

CANCEL_OPTION

OK_OPTION

CLOSED_OPTION
```

Figure 16-13. *JOptionPane constants for the showOptionDialog returned value*

If you specify your own list of button labels, you can use the returned value as an index into the options array to determine the label of the selected button. Be careful, though. Before using it as an index, you should check to see if a value of -1 (the JOptionPane.CLOSED_OPTION) was returned in case the user closed the dialog box.

The JOptionPane.showConfirmDialog() Method

By now, you should be very comfortable with the parameters that exist for the JOptionPane.show*Dialog() methods. This will make the explanation of the showConfirmDialog() method that much easier. The simplest form of this method is shown in Figure 16-14.

```
showConfirmDialog(
    Component parentComponent,
    Object    message
)
```

Figure 16-14. *Simple showConfirmDialog() arguments*

This method can be used to display a simple dialog box with three choices (buttons): Yes, No, and Cancel. This is identical to calling the showOptionDialog() method and specifying an optionType value of JOptionPane.YES_NO_ CANCEL_OPTION. The default title is “Select an Option,” and you can use the JOptionPane constants listed in Figure 16-13 to determine which selection the user chose.

The other showConfirmDialog() signatures follow this same pattern, with one allowing an optionType, the next having an optionType as well as a messageType, and the last allowing these two arguments in addition to an icon. Again, when the optionType is provided, it determines the number of buttons and which values should be shown. The messageType identifies the icon to be

shown, if the icon parameter isn't specified.

The JOptionPane.showInputDialog() Method

The last of the show*Dialog() methods discussed in this chapter is showInputDialog(). It has a number of variations. You'll take a look at these variations so you can decide which make most sense for your needs. Let's start by looking at the simplest one, which has only one parameter and work your way toward the most complex version, which has seven parameters.

Figure 16-15 shows the simplest variant, which requires only a string parameter containing the message to be displayed. In fact, there are two forms of this variant—with and without the parentComponent parameter.

```
| showInputDialog(                                                |
|     Object    message                                           |
| )                                                               |
```

```
| showInputDialog(                                                |
|     Component parentComponent,                                  |
|     Object    message                                           |
| )                                                               |
```

Figure 16-15. *Simplest showInputDialog() method signatures*

Listing 16-12 shows the showID() method from the InputDialogDemo.py application, with the optional parentComponent specified.

Listing 16-12. Simplest showInputDialog() Example

```
26| def showID( self, event ) :
27| result = JOptionPane.showInputDialog(
28| self.frame, # parentComponent
29| 'What is your favorite color?' # message text
30| )
31| self.label.setText( 'result = "%s"' % result )
```

The result of executing the showInputDialog() method call shown in Listing 16-12 is illustrated in Figure 16-16. Personally, I haven't been able to figure out a difference between these two variants. Regardless of whether the optional parentComponent is specified, you see a dialog box with a title of "Input" and the specified message, as well as OK and Cancel buttons.

Figure 16-16. *Sample showInputDialog() window*

One interesting thing to note is that the value that is returned is not an integer, as you have seen with other show*Dialog() methods. It is either the user-specified string (the contents of the input field) or None if the dialog box is closed (which is done by using the Close icon or pressing the Cancel button).

Figure 16-17 shows the next pair of signatures for this method. In it you see that an initial value for the input field can be specified. Again, there are two variants of this showInputDialog() method, with the parentComponent being optional.

```
| showInputDialog(                                                    |
|     Object    message,                                              |
|     Object    initialSelectionValue                                 |
| )                                                                   |
```

```
| showInputDialog(                                                    |
|     Component parentComponent,                                      |
|     Object    message,                                              |
|     Object    initialSelectionValue                                 |
| )                                                                   |
```

Figure 16-17. *showInputDialog() with initialSelectionValue specified*

Listing 16-13 shows an example of this method call from the InputDialogDemo.py sample application. In this particular example, the optional parentComponent is specified in line 28. Feel free to try this version and then comment out that line and try it again.

Listing 16-13. showInputDialog() with an Initial Value

```
26| def showID( self, event ) :
27| result = JOptionPane.showInputDialog(
28| self.frame, # parentComponent
```

```
29| 'What is your favorite color?', # message text
30| 'Spam' # initialValue
31| )
32| self.label.setText( 'result = "%s"' % result )
```

When I tried this, I saw output similar to what is shown in Figure 16-18. Note how the initial value is provided and selected.

Figure 16-18. *Sample showInputDialog() window showing an initial value*

The next variation allows you to specify the title of the dialog box, as well as the messageType parameter that determines the icon to display. The signature for this version is shown in Figure 16-19.

```
| showInputDialog(                                                    |
|     Component parentComponent,                                      |
|     Object    message,                                              |
|     String    title,                                                |
|     int       messageType                                           |
| )                                                                   |
```

Figure 16-19. *showInputDialog() method with messageType parameter*

Listing 16-14 shows how the title and messageType parameters are specified. The constants to be used for the messageType parameter can be found in Figure 16-7.

Listing 16-14. showInputDialog() with Title and messageType Parameters

```
26| def showID( self, event ) :
27| result = JOptionPane.showInputDialog(
28| self.frame, # parentComponent
29| 'What is your favorite color?', # message text
30| 'Asked by the bridge guardian', # title
31| JOptionPane.QUESTION_MESSAGE # messageType
32| )
33| self.label.setText( 'result = "%s"' % result )
```

The output of the version of the showID() method in Listing 16-14 is shown in

Figure 16-20.

Figure 16-20. *Sample output showing the title and messageType*

Finally, you get to the last of the showInputDialog() method variants. As mentioned earlier, its signature is shown in Figure 16-21. The main difference between this method and the other signatures just discussed is the presence of the icon, selectionValues, and initialSelectionValue arguments.

```
| showInputDialog(                                                    |
|     Component parentComponent,                                      |
|     Object    message,                                              |
|     String    title,                                                |
|     int       messageType,                                          |
|     Icon      icon,                                                 |
|     Object[]  selectionValues,                                      |
|     Object    initialSelectionValue                                 |
| )                                                                   |
```

Figure 16-21. *Seven argument showInputDialog() method signature*

Listing 16-15 shows an example of this seven argument variant of the showInputDialog() method.[8] As mentioned, if an icon isn't provided, the icon that's displayed on the MessageDialog is based on the specified messageType value.

[8]As you've seen previously, the lines have been shortened to fit in the available space. This is most obvious with lines 27 and 28, which really don't require two statements.

Listing 16-15. Sample Use of Seven Argument showInputDialog() Method

```
26| def showID( self, event ) :
27| COLORS = 'Red,Orange,Yellow,Green,Blue,Indigo,Violet'
28| colors = COLORS.split( ',' )
29| result = JOptionPane.showInputDialog(
```

```
30| self.frame, # parentComponent
31| 'What is your favorite color?', # message text
32| 'Asked by the bridge guardian', # title
33| JOptionPane.QUESTION_MESSAGE, # messageType
34| None, # icon
35| colors, # selectionValues
36| colors[ -1 ] # initialSelectionValue
37| )
38| self.label.setText( 'result = "%s"' % result )
```

Figure 16-22 shows images from the application containing the method shown in Listing 16-15. This variant of the showInputDialog() method provides a combo box containing the values provided by the selectionValues argument. The last parameter is used to identify the initial combo box entry to be displayed.

Figure 16-22. *Sample output from seven argument showInputDialog() call*

Summary

This chapter has been all about dialog boxes and some of the ways they can be used in your applications. You've seen how simple these techniques can be. The large variety of ways that dialog boxes can be created and used should be a clue about how useful they can be. I encourage you to take a look at the sample scripts in the code\Chap_16 directory and modify them to test the various techniques described in this chapter.

Additionally, you are encouraged to take a look at the "How to Make Dialogs"[9] portion of the Java Swing Tutorial, which contains much of this same information but uses Java instead of Jython as the programming language. One

thing you'll take away from reading that document is how much easier it can be to create and use dialog boxes in Jython.

In the next chapter, you investigate specialized dialog boxes, including how to create and use them in your applications.

[9]See http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html.

Chapter 17

Specialized Dialog Boxes

In Chapter 16, you saw how to create a completely customized dialog box using the JDialog[1] class. You also saw how to take advantage of the JOptionPane[2] methods to quickly and easily display some simple, generalized dialog boxes. Now you are going to look at some dialog boxes that can be used to make your life as a developer significantly easier. You'll be able to use dialog boxes that have been designed, implemented, and tested for you. The chapter begins by looking at the JFileChooser[3] class, which the Swing developers were kind enough to provide for creating dialog boxes. These are great examples of the kind of user input that can be performed by dialog boxes. Keep these in mind when you encounter a situation where you want to provide your own dialog boxes.

The JFileChooser Class

One of the most common types of dialog boxes is the ones that let the users traverse the filesystem and specify a file or directory to be used by the application. It is important to remember, though, that the JFileChooser instance helps your application to interact with the users to choose a particular file or directory. It is the responsibility of the application to do something with the specified file or directory.

Let's take a look at just how difficult it is to use the JFileChooser class. Listing 17-1 shows a simple event handler that displays a File Chooser dialog box that allows the users to traverse the local filesystem and select a file to be opened.

Listing 17-1. Trivial Sample JFileChooser() Routine

```
def showFC( self, event ) :
fc = JFileChooser()
```

```
result = fc.showOpenDialog( None )
if result == JFileChooser.APPROVE_OPTION :
message = 'result = "%s"' % fc.getSelectedFile()
else :
message = 'Request canceled by user'
self.label.setText( message )
```

Figure 17-1 shows an example dialog box after the user has moved about the filesystem; it's displaying the contents of the C:\IBM\WebSphere directory.

[1] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JDialog.html.

[2] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JOptionPane.html.

[3] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFileChooser.html.

Open
Look In:
WebSphere
AppServer70
AppServer80
AppServer85
db2drivers
Education
scripts
UpdateInstaller
Configuring_VMM_to_use_Bluepa
Portzilla 85 AppSrv00.png
Portzilla 85 AppSrv01.png
Portzilla 85 DM.png
Portzilla 85 NA.png
File Name:
Files of Type:
All Files
Open
Cancel

Open
Look In:
WebSphere
Desktop
Computer
Local Disk (C:)
IBM
WebSphere
DVD RW Drive (D:)
Network
Libraries
File Name:
Files of Type:
All Files
Open
Cancel

Figure 17-1. *Sample JFileChooser() Open dialog box from FileChooserDemo1.py*

It doesn't take much to realize how powerful this dialog box can be. For example, clicking on the Look In dropdown shows a combo box with indented entries for each directory level, as well as some specialized icons to indicate directories and disk drives.

Now let's take a look at which constructors are available with the JFileChooser class, so that you can have your applications create the appropriate kind of JFileChooser instance best suited for your needs.
JFileChooser Constructors

Which constructors exist for the JFileChooser class? Table 17-1 shows the Java constructor signatures and includes a short description about each constructor. The first constructor is the one that was used in Listing 17-1, in line 27. The next two signatures allow you to identify the starting view of the JFileChooser dialog box using either java.io.File or java.lang.String to identify the initial view to be shown. That makes it easy to indicate the starting directory to be displayed by the JFileChooser instance.

Table 17-1. *JFileChooser Constructors* **Signature Description**

JFileChooser()
JFileChooser(File currentDirectory) JFileChooser(String currentDirectoryPath) JFileChooser(FileSystemView fsv)

JFileChooser(File currentDirectory, FileSystemView fsv)
JFileChooser(String currentDirectoryPath, FileSystemView fsv)

Constructs a JFileChooser pointing to the user's default directory. Constructs a JFileChooser using the given File as the path. Constructs a JFileChooser using the given Path.
Constructs a JFileChooser using the given FileSystemView.

Constructs a JFileChooser using the given currentDirectory and FileSystemView.
Constructs a JFileChooser using the given currentDirectoryPath and FileSystemView.

The last three JFileChooser constructors include a FileSystemView parameter

that can be used to limit the portion of the filesystem that the JFileChooser can display to the user. This way, you don't have to allow the user to see the whole filesystem, just the portion they need.

Using a FileSystemView

The default FileSystemView instance that is used by the JFileChooser allows you to access the complete filesystem. Listing 17-2 shows a verbose way to provide or identify the default FileSystemView instance. It is, in fact, identical to the default JFileChooser( ) instantiation when no parameter is specified.

Listing 17-2. JFileChooser( ) Instance with Default FileSystemView from FileChooserDemo2.py

```
27| def showFC( self, event ) :
28| fc = JFileChooser( FileSystemView.getFileSystemView() )
29| result = fc.showOpenDialog( None )
30| if result == JFileChooser.APPROVE_OPTION :
31| message = 'result = "%s"' % fc.getSelectedFile()
32| else :
33| message = 'Request canceled by user'
34| self.label.setText( message )
```

If, on the other hand, you are interested in limiting the user to a specific subset of the filesystem, you can implement a class that identifies the virtual "root" of the filesystem that you want the user to be able to traverse. Listing 17-3 shows one implementation of this kind of FileSystemView descendent class.[4]

Listing 17-3. FileSystemView Class that Limits Filesystem Access from FileChooserDemo3.py

```
11|class RestrictedFileSystemView( FileSystemView ) :
12| def __init__( self, root ) :
13| FileSystemView.__init__( self )
14| self.root = root
15| self.Roots = [ root ]
16| def createNewFolder( self, containingDir ) :
17| folder = File( containingDir, 'New Folder' )
18| folder.mkdir()
19| return folder
```

```
def getDefaultDirectory( self ) :
return self.root
def getHomeDirectory( self ) :
return self.root
def getRoots( self ) :
return self.Roots
```

Listing 17-4 shows how this RestrictedFileSystemView class might be used to limit the JFileChooser to the specified directory and its subdirectories. It is unlikely, however, that you are going to want the “root” directory specified using a string like this one. It would be more reasonable to include statements to determine the root system based on the specific environment in which the script is executing.

Listing 17-4. JFileChooser() Using RestrictedFileSystemView from FileChooserDemo3.py

```
def showFC( self, event ) :
fc = JFileChooser(
RestrictedFileSystemView(
File( r'C:\IBM\WebSphere' )
)
)
result = fc.showOpenDialog( None )
if result == JFileChooser.APPROVE_OPTION :
message = 'result = "%s"' % fc.getSelectedFile()
else :
message = 'Request canceled by user'
self.label.setText( message )
```

Figure 17-2 shows some sample images from the application from Listings 17-3 and 17-4. It is interesting to see how the initial directory is WebSphere, and that the “up one level” button is disabled because WebSphere is considered the “root” directory of this restricted filesystem.

[4]Note: The FileSystemView roots attribute is read-only, so this class works around that limitation by using a variable named Roots instead.

Open
Look In:
WebSphere
AppServer70
AppServer80
AppServer85
db2drivers
Education
scripts
UpdateInstaller
Configuring_VMM_to_use_Bluepa
Portzilla 85 AppSrv00.png
Portzilla 85 AppSrv01.png
Portzilla 85 DM.png
Portzilla 85 NA.png
File Name:
Files of Type:
All Files
Open
Cancel

Open
Look In:
WebSphere
WebSphere
AppServer80
AppServer85
db2drivers
Education
scripts
Configuring_VMM_to_use_Bluepa
Portzilla 85 AppSrv00.png
Portzilla 85 AppSrv01.png
Portzilla 85 DM.png
Portzilla 85 NA.png
File Name:
Files of Type:
All Files
Open
Cancel

Figure 17-2. *Sample RestrictedFileSystem JFileChooser() images*

Does this FileSystemViewer class completely limit the users from accessing the rest of the system? No, not really. For example, if the user enters something like C:\ in the File Name input field, the view will display the contents of that directory, unfortunately. There may be a way to add an input verifier to the JFileChooser input field to intercept this kind of thing, but I haven't investigated that option. If you have some success with that approach, please let me know.

File Filtering

By default, a JFileChooser instance will show all of the (non-hidden) files and directories to the user. There are times when you might prefer to limit the kinds of files to be displayed. For example, you might want to allow the user to see only the XML, text, or image files. The way to do this is by adding one or more FileFilter[5] instances to the JFileChooser. The JFileChooser then determines which files and directories should be visible.

One of the most common FileFilter mechanisms is to identify the viewable files based on file extension. To simplify this process, the Swing developers have provided a FileNameExtensionFilter[6] class.
Listing 17-5 shows how easily you can add multiple FileNameExtensionFilter instances to a file chooser. This allows users to select the kinds of files to be

shown based on the filename extensions.

Listing 17-5. The showFC Method from the FileChooserDemo4.py Script

```
44| def showFC( self, event ) :
45| fc = JFileChooser(
46| RestrictedFileSystemView(
47| File( r'C:\IBM\WebSphere' )
48| )
49| )
50| fc.addChoosableFileFilter(
51| FileNameExtensionFilter(
52| 'XML files',
53| [ 'xml' ]
54| )
55| )
56| fc.addChoosableFileFilter(
57| FileNameExtensionFilter(
58| 'Image files',
59| 'bmp,jpg,jpeg,gif,png'.split( ',' )
60| )
61| )
62| fc.addChoosableFileFilter(
63| FileNameExtensionFilter(
64| 'Text files',
65| [ 'txt' ]
66| )
67| )
68| result = fc.showOpenDialog( None )
69| if result == JFileChooser.APPROVE_OPTION :
70| message = 'result = "%s"' % fc.getSelectedFile()
71| else :
72| message = 'Request canceled by user'
73| self.label.setText( message )
```

Figure 17-3 shows an example image from the FileChooserDemo4.py script. It shows the filters available when the Files of Type combo box is selected. It is interesting to note that the initial filter to be displayed is the last one that was added. It is also interesting to note that the All Files filter is available by default. So, if you don’t want this filter to be available, you need to use the

removeChoosableFileFilter() method and pass it the result of calling the getAcceptAllFileFilter(). The FileChooserDemo5.py script includes these calls, in case you are interested.

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/filechooser/FileFilter.html.
[6]See http://docs.oracle.com/javase/8/docs/api/javax/swing/filechooser/FileNameExtens

Figure 17-3. *Sample output from the FileChooserDemo4.py script*

Chooser Dialog Types

Up to now, all of the examples that you've seen have used the showOpenDialog() method to display the modal dialog box. In each of these, the primary button displayed the Open text. There are two other ways to display the dialog box. The first uses the showSaveDialog() method, which includes a Save button. This shouldn't be too much of a surprise. The second allows you to customize the text to be displayed on the primary button and uses the showDialog() method to do so. In each case, the application uses the return value

to determine which choice the user made so that the appropriate action can be taken. The possible return values are:

• JFileChooser.APPROVE_OPTION
• JFileChooser.CANCEL_OPTION
• JFileChooser.ERROR_OPTION

The last of these occurs only if some kind of error is encountered or if the dialog is somehow dismissed.
Selection Types

There are times when you'll want your users to be able to choose something other than a file. For these times, you need to tell the JFileChooser instance the kinds of selections that are acceptable. To do this, you can use the setFileSelectionMode() method or the fileSelectionMode attribute keyword on the constructor call. In either case, the values that can be specified are as follows:

• JFileChooser.FILES_ONLY
• JFileChooser.DIRECTORIES_ONLY
• JFileChooser.FILES_AND_DIRECTORIES

The default value is JFileChooser.FILES_ONLY, which shouldn't be much of a surprise. You might wonder why the primary button isn't disabled when this in effect. I was a little surprised about this myself, until I selected a directory and clicked on the Open button. The JFileChooser understood this to be the same as double-clicking on the directory, so I guess it does make some sense.

You might want to use the JFileChooser.DIRECTORIES_ONLY value for a chooser that allows the user to identify a source or destination directory for some operation (such as for copying files). There are a number of other JFileChooser methods as well. You might want to take a look at the Swing Tutorial page entitled "How to Use File Choosers"[7] if you are interested.

The JColorChooser Class

Another specialized dialog box that is included in the Swing hierarchy is the JColorChooser[8] class. It allows users to display a variety of techniques that can be used to select a color. A trivial script can be used to display a modal color

chooser dialog box. Figure 17-4 shows some views from this dialog box.

[7]See http://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html.
[8]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JColorChooser.html.

Figure 17-4. *Sample output from the ColorChooserDemo1.py script*

Listing 17-6 shows the showCC ActionListener event handler method from the ColorChooserDemo1.py script used to generate the output shown in Figure 17-4.

Listing 17-6. The showCC Method from the ColorChooserDemo1.py Script

```
27| def showCC( self, event ) :
28| result = JColorChooser().showDialog(
29| None, # Parent component
```

```
30| 'Color Selection', # Dialog title
31| self.label.getForeground() # Initial color
32| )
33| if result :
34| message = 'New color: "%s"' % result.toString() 35| self.label.setForeground(
result )
36| else :
37| message = 'Request canceled by user' 38| self.label.setText( message )
```

One of the interesting things about this color chooser dialog is the fact that the user has multiple ways of making a selection. As you can see in Figure 17-4, there are three tabs, any of which can be used by the user to pick a color. Aren't you glad that the Swing developers have gone through the effort of creating this? It is certainly much easier to use this than to create your own.

The javax.swing.colorchooser Package

In order to provide this functionality, the JColorChooser uses some support classes in the javax.swing.colorchooser package. You can determine this by taking a look at the JColorChooser constructors, as shown in Table 17-2. The first two constructors in this table are pretty straightforward.

Table 17-2. *JColorChooser Constructors* **Signature Description**
JColorChooser( )
JColorChooser(Color initialColor)

JColorChooser
(ColorSelectionModel model) Creates a color chooser pane with a default color of white.

Creates a color chooser pane with the user-specified initial color. Creates a color chooser pane with the given ColorSelectionModel.
Paranoid developers might wonder if the default color of the first, no-argument constructor is in fact white. In fact, you can easily verify this using an expression like the one shown on lines 5 and 6 of Listing 17-7.
Listing 17-7. Verifing the Default JColorChooser Color

```
1|wsadmin>from java.awt import Color
2|wsadmin>from javax.swing import JColorChooser
3|wsadmin>
```

```
4|wsadmin>cc = JColorChooser()
5|wsadmin>cc.getColor() == Color.white
6|1
7|wsadmin>
8|wsadmin>csm = cc.getSelectionModel()
9|wsadmin>print csm.toString().split( '.' )[ -1 ]
10|DefaultColorSelectionModel@768b768b
11|wsadmin>
```

You might wonder about the last of the constructors shown in Table 17-2. What is a ColorSelectionModel?[9] The Javadoc says that it is an interface, and not a simple class. That same page says that the DefaultColorSelectionModel[10] class is the only implementation class that is provided by in the Swing hierarchy. In lines 8-10 of Figure 17-5, you can see that the default ColorSelectionModel that is associated with a JColorChooser instance is one of these DefaultColorSelectionModel instances.

Figure 17-5. *JColorChooser sample output after making a change*

Looking at the Javadoc for the DefaultColorSelectionModel class, you see that it allows you to listen for changes to the color. As you have seen before, this is the standard way that can be monitored. From this experience, you should be able to make an educated "guess" as to how the JColorChooser works. Take another look at the

[9]See http://docs.oracle.com/javase/8/docs/api/javax/swing/colorchooser/ColorSelectio
[10]See http://docs.oracle.com/javase/8/docs/api/javax/swing/colorchooser/DefaultColorS

ColorChooserDemo1.py sample script. What is the initial color that is displayed? Take a look at each of the tabs and you should see that the same color shown in the preview pane is represented on each tab in a consistent fashion. What happens when you pick another color and then select the other tabs? How do you think that this is done? Figure 17-5 shows some sample images after one color change has been made. You are encouraged to test this application yourself to see the JColorChooser in full size.

What Else Do You Need to Know About These "Special" Dialog Boxes?

One thing that may be useful to understand about these classes is the fact that they are based on the javax.swing. JComponent class, as you can see in Listing 17-8. What does this mean? The interesting thing is that since they are both JComponents, they aren't limited to being used as modal dialog boxes. You can, if you want, add an instance of either to any container, just like you can for any JComponent. You don't have to, but it's something to consider.

Listing 17-8. JColorChooser and JFileChooser Class Hierarchy

```
wsadmin>from javax.swing import JColorChooser wsadmin>from javax.swing
import JFileChooser wsadmin>
wsadmin>classInfo( JColorChooser )
javax.swing.JColorChooser
| javax.swing.JComponent
| | java.awt.Container
| | | java.awt.Component
| | | | java.lang.Object
| | | | java.awt.image.ImageObserver
| | | | java.awt.MenuContainer
| | | | java.io.Serializable
| | java.io.Serializable
| javax.accessibility.Accessible
wsadmin>
wsadmin>classInfo( JFileChooser )
javax.swing.JFileChooser
| javax.swing.JComponent
| | java.awt.Container
| | | java.awt.Component
```

```
| | | | java.lang.Object
| | | | java.awt.image.ImageObserver
| | | | java.awt.MenuContainer
| | | | java.io.Serializable
| | java.io.Serializable
| javax.accessibility.Accessible
wsadmin>
```

Summary

This chapter covered some specialized dialog boxes that are highly functional and generally useful all at the same time. The ones you learned about here are related to user selection—selecting a color, a file, or a directory. Hopefully, you have seen just how easily they can be added to your scripts. Coming up in the next chapter, you'll learn how your applications can monitor progress and report it back to the users.

Chapter 18

Monitoring and Indicating Progress

As you've no doubt seen, many graphical applications include a progress indicator of some sort to convey how quickly something is happening. I don't know about you, but I tend to be somewhat impatient, so I find these progress bars very helpful. I hate it when a program just sits there—is it still working, has it hung, who knows? This chapter covers some different ways to measure progress and communicate it to your users.

Changing the Cursor

Sometimes you'll need to tell the users that the program is busy and let them know that they have to wait for a (hopefully short) amount of time. The easiest way to do this is to change the Cursor[1] from its default value to one that indicates that the application is busy. The Cursor class includes a number of predefined constants that can be used for this purpose. For this particular situation, you can use WAIT_CURSOR. How do you do that? Listing 18-1 shows a basic example of how to use it.

Listing 18-1. Using the WAIT_CURSOR Method from WaitCursor1.py

```
23| def wait( self, event ) :
24| source = event.getSource()
25| prev = source.getCursor()
26| source.setCursor(
27| Cursor.getPredefinedCursor( Cursor.WAIT_CURSOR )
28| )
29| sleep( 5 )
30| source.setCursor( prev )
```

This method is from the WaitCursor1.py script in the code\Chap_18 directory. If you execute the script and press the Wait button, the cursor will change from its default value to the WAIT_CURSOR for five seconds. Remember to change it back. Otherwise, you are likely to confuse your users into thinking that the script is still busy. Figure 18-1 shows some sample images from my system when I executed this script.

[1]Seehttp://docs.oracle.com/javase/8/docs/api/java/awt/Cursor.html.

Figure 18-1. *WaitCursor images from a Windows 7 environment*

■ **Note** it is important to note that the way that the cursor looks is dependent on the operating system. it's also important to remember that WAIT_CURSOR is visible only when the cursor is over the component for which the cursor was changed. Because of this, you may want to call the setCursor() method for your highest-level container (for example, your frame instance).

What if you want to enable WAIT_CURSOR in one part of your application and change it back somewhere else? You either have to save the original cursor setting somewhere, so it can be restored using this saved value, or you can simply use the Cursor.DEFAULT_CURSOR constant. Listing 18-2 demonstrates

this constant using a JToggleButton. An even easier technique is to use None as an argument to the setCursor(...) method, which will force the specified component to use the cursor setting of its parent component.

Listing 18-2. Setting the Cursor Shape Based on isSelected() from WaitCursor2.py

```
22| def wait( self, event ) :
23| source = event.getSource()
24| cursor = [
25| Cursor.DEFAULT_CURSOR, # isSelected() == 0 (false)
26| Cursor.WAIT_CURSOR # isSelected() == 1 (true)
27| ][ source.isSelected() ]
28| source.setCursor( Cursor.getPredefinedCursor( cursor ) )
```

When the ActionListener event handler is called, the cursor state will be based on the result of calling the JToggleButton isSelected() method (which returns zero for false and one for true).
Are there any problems with using the cursor to indicate that the application is busy? Think about it. When you use this technique, you are telling the user to wait. How happy do you think the users are when they are told to wait? I, for one, hate when I get a “wait” message. Since your applications are supposed to be event driven, you want them to be able to continue running even though a different part might be busy doing something. If your script needs to be interacting with the user while another part is doing something else, it needs to be doing these other operations on separate threads. Remember the SwingWorker class first mentioned in Chapter 6? That’s how you’re going to do it.

Showing a Progress Bar

Have you ever had an application appear to stop working and wonder what was going on? It happens to me all of the time. Even though I have a fairly capable laptop, there are times when every application appears to be hung. Sometimes the title bars show a “(Not Responding)” message, but that isn’t very reassuring. Unless an operation is expected to complete in less than half a second, it is best to consider communicating to the user some kind of indication of the expected wait time. That’s the purpose of a progress bar. Let’s take a look at the JProgressBar[2] class and see what it takes to make use of it. It’s always a good

idea to begin by looking at the class constructors. Table 18-1 contains the constructors for the JProgressBar class.

Table 18-1. *JProgressBar Constructors* **Signature Description** JProgressBar()
JProgressBar(BoundedRangeModel newModel)
JProgressBar(int orient)
JProgressBar(int min, int max)

JProgressBar(int orient, int min, int max) Creates a horizontal progress bar that displays a border but no progress string.

Creates a horizontal progress bar that uses the specified model to hold the progress bar's data.

Creates a progress bar with the specified orientation, which can be SwingConstants.VERTICAL or SwingConstants. HORIZONTAL.

Creates a horizontal progress bar with the specified minimum and maximum. Creates a progress bar using the specified orientation, minimum, and maximum.

As is often the case, at least for me, the description that is provided in the Javadoc isn't always clear. Take a look at the default JProgressBar instance. The first implementation can be found in ProgressBar0.py; sample images from this implementation are shown in Figure 18-2.

Figure 18-2. *Sample images from the ProgressBar0.py output*

[2]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JProgressBar.html.

How was this done? The primary (ProgressBar) class for this script is shown in Listing 18-3. There really shouldn't be too many surprises here. The only part that you might wonder about is using the setBorder(...) method call on lines 46-48.

It's sometimes confusing that you have to use the frame.getContentPane() method to access the particular pane to which you want to add an "empty" border. Note also the call to the BorderFactory[3].createEmptyBorder(...) method to create a little space or gap around the components in the JPanel. You might want to take a look at the "How to Use Borders" section of the Java Swing Tutorials[4] to obtain a better understanding about borders and how they can be used in Swing applications.

Listing 18-3. ProgressBar Class from ProgressBar0.py

```
38|class ProgressBar0( java.lang.Runnable ) :
39| def run( self ) :
40| frame = JFrame(
41| 'ProgressBar0',
42| size = ( 280, 125 ),
43| locationRelativeTo = None,
44| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
45| )
46| frame.getContentPane().setBorder(
47| BorderFactory.createEmptyBorder( 20, 20, 20, 20 )
48| )
49| panel = JPanel()
50| self.button = panel.add(
51| JButton(
52| 'Start',
53| actionPerformed = self.start
54| )
55| )
56| self.progressBar = panel.add( JProgressBar() )
57| frame.add(
58| panel,
59| BorderLayout.NORTH
60| )
61| frame.setVisible( 1 )
```

```
62| def start( self, event ) :
63| progressTask(
64| self.button,
65| self.progressBar
66| ).execute()
```

The ActionListener event handler routine, shown in lines 62-66 of Listing 18-3, creates the progressTask instance. Immediately after that, the thread starts executing. Details about this class are shown in Listing 18-4.

[3]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/BorderFactory.html.

[4]See http://docs.oracle.com/javase/tutorial/uiswing/components/border.html.

Listing 18-4. The progressTask class from ProgressBar0.py

```
14|class progressTask( SwingWorker ) :
15| def __init__( self, button, progressBar ) :
16| self.btn = button # Save provided references
17| self.PB = progressBar
18| SwingWorker.__init__( self )
19| def doInBackground( self ) :
20| self.btn.setEnabled( 0 ) # Disable the "start" button
21| try :
22| random = Random()
23| progress = 0
24| self.PB.setValue( progress )
25| while progress < 100 :
26| sleep( ( random.nextInt( 1400 ) + 100 ) / 1000.0 )
27| progress = min(
28| progress + random.nextInt( 10 ) + 1, 100
29| )
30| self.PB.setValue( progress )
31| except :
32| Type, value = sys.exc_info()[ :2 ]
33| print 'Error:', str( Type )
34| print 'value:', str( value )
35| sys.exit()
36| def done( self ) :
37| self.btn.setEnabled( 1 ) # Enable the "start" button
```

Is this a great example of a SwingWorker descendent class? No, not really. For

one thing, it needs to know too much about the invoking application. This should be obvious when you see that the constructor needs to have references to specific application components provided when the object is instantiated. This should be a dead giveaway. There has to be a better way, don't you think?

SwingWorker Progress

Are there any methods in the SwingWorker class related to "progress"? How can you find out? Chapter 4, introduced the classInfo utility class. Listing 18-5 shows the output of this function. It tells you that a getter method exists for a property named progress. Looking at the Javadoc for the SwingWorker[5] class shows you that a setter method exists, but it is protected, so any descendent classes that you define have to do something special to access that setter method.

Listing 18-5. SwingWorker "Progress" Methods

```
wsadmin>from javax.swing import SwingWorker wsadmin>
wsadmin>classInfo( SwingWorker, meth = 'progress' )
javax.swing.SwingWorker

getProgress
| java.lang.Object
| java.util.concurrent.RunnableFuture | | java.lang.Runnable
| | java.util.concurrent.Future wsadmin>
```

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/SwingWorker.html.

How do I know this? When I tried to use a simple call to the SwingWorker setProgress(...) method, an AttributeError exception was raised and the setProgress name was the source of the error.[6]
So, what can you do? The Jython developers were kind enough to provide a way to access this kind of protected method. A Java programmer can use the @Override annotation to allow the doInBackground(...) method to use the setProgress(...) method. In Jython, you have to use the following syntax to call the setProgress(...) method. In the SwingWorker descendent class, use this:

self.super__setProgress(value)[7]

Listing 18-6 shows the modified progressTask class from the working

ProgressBar2.py script. Note the simplifications, which include the fact that this class now has no references to the application components. That means it's completely contained, meaning that all references to the class attributes exist only in the class.

Listing 18-6. The progressTask Class from ProgressBar2.py

```
15|class progressTask( SwingWorker ) :
16| def __init__( self ) :
17| SwingWorker.__init__( self )
18| def doInBackground( self ) :
19| try :
20| random = Random()
21| progress = 0
22| self.super__setProgress( progress )
23| while progress < 100 :
24| sleep( ( random.nextInt( 1400 ) + 100 ) / 1000.0 )
25| progress = min(
26| progress + random.nextInt( 10 ) + 1, 100
27| )
28| self.super__setProgress( progress )
29| except :
30| Type, value = sys.exc_info()[ :2 ]
31| print 'Error:', str( Type )
32| print 'value:', str( value )
33| sys.exit()
34| def done( self ) :
35| pass
```

Listing 18-7 shows the modified ProgressBar class from this same script. Since the SwingWorker descendent class knows nothing about the class components defined here, you have to make this class a descendent of the PropertyChangeListener class, as shown on lines 36 and 65-70.

[6]The script containing the "first attempt" is the ProgressBar1.py file found in the code\Chap_18 directory. [7]Yes, there really are two underscores between super and setProgress.

Listing 18-7. The ProgressBar Class from ProgressBar2.py

```
36|class ProgressBar2( java.lang.Runnable, PropertyChangeListener ) :
37| def run( self ) :
38| frame = JFrame(
39| 'ProgressBar2',
40| size = ( 280, 125 ),
41| locationRelativeTo = None,
42| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
43| )
44| frame.getContentPane().setBorder(
45| BorderFactory.createEmptyBorder( 20, 20, 20, 20 )
46| )
47| panel = JPanel()
48| self.button = panel.add(
49| JButton(
50| 'Start',
51| actionPerformed = self.start
52| )
53| )
54| self.progressBar = panel.add( JProgressBar() )
55| frame.add(
56| panel,
57| BorderLayout.NORTH
58| )
59| frame.setVisible( 1 )
60| def start( self, event ) :
61| self.button.setEnabled( 0 )
62| task = progressTask()
63| task.addPropertyChangeListener( self )
64| task.execute()
65| def propertyChange( self, event ) :
66| if event.getPropertyName() == 'progress' :
67| progress = event.getNewValue()
68| self.progressBar.setValue( progress )
69| if progress == 100 :
70| self.button.setEnabled( 1 )
```

Showing Progress Details

So far, this is a pretty good way to display progress, albeit a bit vague. Wouldn’t

it be nice if you were able to display the percentage complete on the progress bar? The developers of the Swing classes thought about this, and have made it really easy for you to do. All you have to do is enable the stringPainted property of the JProgressBar instance. In Jython, this is as simple as adding the stringPainted keyword to the JProgressBar constructor call. Listing 18-8 shows this process.

Listing 18-8. ProgressBar Constructor Call with the stringPainted Keyword

```
56| self.progressBar = panel.add(
57| JProgressBar( stringPainted = 1 )
58| )
```

After making this change, you can see that the progress bar includes a string representation of the completion as a percentage value. Figure 18-3 shows some sample images from ProgressBar4.py, which incorporates this change.

Figure 18-3. *Sample images with stringPainted enabled from ProgressBar4.py*

Specifying a Progress Bar Range

Some of the JProgressBar constructors shown in Table 18-1 allow you to define the minimum and maximum values for your progress bar. What does this do, and what does this mean as far as the SwingWorker progress?

If you specify minimum and maximum values for a progress bar, you become responsible for tracking its updates. The separation of the progressTask (SwingWorker) instance from the ProgressBar class means that only the progress bar is aware of the new minimum and maximum values. In the progressTask instance, it is measuring a percentage of the progress, as shown in Figure 18-3.

So, if your application needs to use non-default minimum and maximum values in a progress bar instance, it needs to take this into account. Listing 18-9 shows one approach that you might want to consider using. Regardless of the minimum and maximum values you choose for the JProgressBar instance, it uses the progress value of the progressTask instance as a percentage complete value to determine the value that's assigned to the JProgressBar instance (as shown on line 74).

Listing 18-9. propertyUpdate() Method from ProgressBar5.py

```
68| def propertyUpdate( self, event ) :
69| if event.getPropertyName() == 'progress' :
70| progress = event.getNewValue() # integer % complete
71| lo = self.progressBar.getMinimum()
72| hi = self.progressBar.getMaximum()
73| here = int( ( hi - lo ) * 0.01 * progress ) + lo
74| self.progressBar.setValue( here )
75| if progress == 100 :
76| self.button.setEnabled( 1 )
```

Indeterminate ProgressBar Range

Sometimes, you won't know how long an action will take. For example, if your application needs to communicate with a remote host to transfer data from there to here, there is likely to be some delay while communication is being established. Once that has occurred, the amount of data can be provided before the copying initiates. While this is happening, it is considered good practice to tell the users that something is happening.

For this purpose, the JProgressBar class includes an indeterminate attribute. When this attribute[8] is enabled, the progress bar will show movement, but no change in progress completion percentage. The ProgressBar6. py sample script shows one way that this attribute can be used. It includes a random delay in the progressTask doInBackground(...) method to simulate some kind of initialization delay before progress changes occur. Figure 18-4 shows some images of the progress bar when the indeterminate attribute is true.

Figure 18-4. *Sample images of indeterminate progress*

Listing 18-10 shows the lines in ProgressBar6.py that relate to using the indeterminate attribute in your application progress bar instance. In the ActionListener event handler invoked when the Start button is selected, you see how the indeterminate attribute is enabled. Then, in the PropertyChangeListener event handler in line 74, you can see how the state of the indeterminate attribute is disabled and the actual progress is determined.

Listing 18-10. Indeterminate Related Code from ProgressBar6.py

```
65| def start( self, event ) :
66| self.button.setEnabled( 0 )
67| self.progressBar.setIndeterminate( 1 )
68| task = progressTask( self.propertyUpdate )
69| task.execute()
70| def propertyUpdate( self, event ) :
71| if event.getPropertyName() == 'progress' :
72| progress = event.getNewValue() # integer % complete
73| PB = self.progressBar
74| PB.setIndeterminate( 0 )
75| lo = PB.getMinimum()
76| hi = PB.getMaximum()
77| here = int( ( hi - lo ) * 0.01 * progress ) + lo
78| PB.setValue( here )
79| if progress == 100 :
80| self.button.setEnabled( 1 )
```

[8] Interestingly enough, the JProgressBar Javadoc doesn't explicitly identify indeterminate as an actual attribute or field. But if you use the classInfo function from Chapter 4, you can see that it does exist and your Jython scripts can access it directly, even though this practice is discouraged.

ProgressMonitor Objects

In many ways, instances of the ProgressMonitor[9] class appear to be similar to the dialog boxes described in Chapters 16 and 17. This is a bit misleading, though. Listing 18-11 shows that the ProgressMonitor class is a descendent of the java.lang.Object class. JDialog descends from the java.awt.Dialog class, and JColorChooser and JFileChooser both descend from the javax.swing.JComponent class, as you learned in Chapter 17.

Listing 18-11. ProgressMonitor Class Hierarchy

```
wsadmin>from javax.swing import ProgressMonitor wsadmin>
wsadmin>classInfo( ProgressMonitor )
javax.swing.ProgressMonitor
| java.lang.Object
| javax.accessibility.Accessible
wsadmin>
```

What does this mean for your applications? For one, ProgressMonitor instances are going to act very different from descendants of the JComponent class. Additionally, you can't add a ProgressMonitor instance to a Swing container. This provides additional information as to why this dialogue-like box is being discussed here, instead of in Chapter 17. How do you use a ProgressMonitor object? Let's begin with the constructor and then discuss how the instance can and should be used. Take a look at the ProgressMonitor constructors shown in Table 18-2.

Table 18-2. *ProgressMonitor Constructor*

Signature	Description
ProgressMonitor(Component parentComponent, Object message, String note, int min, int max)	Instantiates an object for the purpose of showing progress.

It’s kind of interesting to see that this class has only one constructor. Most of the previous Swing classes you’ve learned about had many more. The constructor parameters shouldn’t be too much of a surprise. The kinds of questions that I had when I first encountered this constructor were mostly concerned with the message and note arguments. Mostly, I wondered about how they differ and how they should be used.

Listing 18-12 shows how simple it can be to demonstrate a progress monitor. If you want to enter this example in your own interactive session, remember that the last (i.e., empty) line is significant. It tells the Jython interpreter that the while statement is complete, and that the contents of the loop should be executed.

[9]See http://docs.oracle.com/javase/8/docs/api/javax/swing/ProgressMonitor.html.

Listing 18-12. Interactive wsadmin Session Showing ProgressMonitor

```
wsadmin>from javax.swing import ProgressMonitor
wsadmin>from java.util import Random
wsadmin>from time import sleep
wsadmin>
wsadmin>random = Random()
wsadmin>progress = 0
wsadmin>
wsadmin>pm = ProgressMonitor( None, 'Message text', None, 0, 100 )
wsadmin>while progress < 100 :
wsadmin> sleep( ( random.nextInt( 1400 ) + 100 ) / 1000.0 ) wsadmin> progress
= min( 100, progress + random.nextInt( 10 ) ) wsadmin> pm.setProgress(
progress )
wsadmin>
```

Figure 18-5 shows some images of a ProgressMonitor instance.

Figure 18-5. *Images from the interactive session ProgressMonitor instance in Listing 18-12*

Note aProgressMonitor instance shouldn't be reused. a new instance should be instantiated instead.

From these, you can see the following about the dialog box:
• A title of Progress...
• A Close icon in the upper-right corner
• An Information icon
• The user-supplied message text
• A progress bar
• A Cancel (or local equivalent) button

Based on this first attempt and on the previous investigation into JDialog instances, you may be surprised to learn that you have no control over:

• The title of the ProgressMonitor dialog box.
• The icon that is displayed
• The buttons that are displayed
• The text that is shown on the button

Some other big differences between ProgressMonitor instances and the other dialog boxes that you've seen are:
• ProgressMonitor dialog box instances are not modal.
• The ProgressMonitor class doesn't have any listeners associated with it.

You may be wondering why I used an interactive wsadmin session, as shown in Listing 18-12, to demonstrate the ProgressMonitor class. One big reason was to show that once the progress reaches the maximum value (100%), the dialog is automatically hidden.

What is required to create a simple script that demonstrates the ProgressMonitor class? Listing 18-13 shows the class that is defined in ProgressMonitor1.py.[10] Notice how the ProgressMonitor is instantiated in the ActionListener event handler, that is, in the start(.) method in lines 65-75.

Listing 18-13. The ProgressMonitor1 Class from ProgressMonitor1.py

```
42|class ProgressMonitor1( java.lang.Runnable ) :
43| def run( self ) :
44| frame = JFrame(
45| 'ProgressMonitor',
46| size = ( 280, 125 ),
47| locationRelativeTo = None,
48| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
49| )
50| frame.getContentPane().setBorder(
51| BorderFactory.createEmptyBorder( 20, 20, 20, 20 )
52| )
53| panel = JPanel()
54| self.button = panel.add(
55| JButton(
56| 'Start',
57| actionPerformed = self.start
58| )
59| )
60| frame.add(
61| panel,
62| BorderLayout.NORTH
63| )
64| frame.setVisible( 1 )
65| def start( self, event ) :
66| self.button.setEnabled( 0 )
67| self.PM = ProgressMonitor(
68| None, # parentComponent 69| 'Message text', # message
```

```
70| None, # note
71| 0, # minimum value 72| 100 # maximum value 73| )
74| task = progressTask( self.propertyUpdate ) 75| task.execute()
```

[10]Note that the class name can't be the same as the Java Swing class you are trying to use.

Listing 18-14 shows the PropertyChangeListener event handler that is defined in the propertyUpdate() method. Note its similarity to the PropertyChangeListener event handler in the progress bar scripts.

Listing 18-14. PropertyChangeListener from ProgressMonitor1.py

```
76| def propertyUpdate( self, event ) :
77| if event.getPropertyName() == 'progress' :
78| progress = event.getNewValue() # integer % complete
79| PM = self.PM
80| lo = PM.getMinimum()
81| hi = PM.getMaximum()
82| here = int( ( hi - lo ) * 0.01 * progress ) + lo
83| PM.setProgress( here )
84| if progress == 100 :
85| self.button.setEnabled( 1 )
```

ProgressMonitor Cancellation

Using ProgressMonitor1.py, you can easily demonstrate the cancellation process. Start the application and press the Start button. Watch what happens when the ProgressMonitor dialog appears. You shouldn't be too surprised that the dialog box goes away. You might wonder why the Start button is still disabled ... at least for a short time.

What is happening? Consider that, when the ProgressMonitor was canceled, the progressTask continued to execute on the separate thread. Until this task completes, PropertyChangeEvents will continue to be generated, which will result in the PropertyChangeListener method being called. Once the progress reaches 100, the Start button will be enabled. Is this the right way to handle the cancellation event? I don't think so.

In order to properly react to the cancellation of a ProgressMonitor, you need to be able to determine that it has occurred. Fortunately, the Swing designers have

provided an isCanceled() method as part of the ProgressMonitor API. You can use it to detect the situation. What should you do when it occurs? Well, one of the important things that you should do is terminate the progressTask that was created to perform the job being monitored. Again, the Swing designers come to the rescue. The SwingWorker API includes a cancel() method that allows developers to terminate the thread.

Is that all you need to worry about? No, you also have to include code in the progressTask class and the PropertyChangeListener event handler to deal with the possible cancellation of these different objects.
Listing 18-15 shows the modifications found in the second script—ProgressMonitor2.py—to deal with the cancellation of a ProgressMonitor object.
Listing 18-15. Differences Found in ProgressMonitor2.py

```
14|class progressTask( SwingWorker ) :
| ...
23| def doInBackground( self ) :
24| try :
25| random = Random()
26| progress = 0
27| self.super__setProgress( progress )
28| sleep( ( random.nextInt( 1400 ) + 100 ) / 1000.0 )
29| while progress < 100 :
30| sleep( ( random.nextInt( 1400 ) + 100 ) / 1000.0 )
31| progress = min(
32| progress + random.nextInt( 10 ) + 1, 100
33| )
34| self.super__setProgress( progress )
35| except KeyboardInterrupt, ki :
36| pass
37| except :
38| Type, value = sys.exc_info()[ :2 ]
39| print 'Error:', str( Type )
40| print 'value:', str( value )
41| sys.exit()
| ...
44|class ProgressMonitor2( java.lang.Runnable ) :
| ...
67| def start( self, event ) :
```

```
68| self.button.setEnabled( 0 )
69| self.PM = ProgressMonitor(
70| None, # parentComponent
71| 'Message text', # message
72| None, # note
73| 0, # minimum value
74| 100 # maximum value
75| )
76| self.task = progressTask( self.propertyUpdate )
77| self.task.execute()
78| def propertyUpdate( self, event ) :
79| if event.getPropertyName() == 'progress' :
80| progress = event.getNewValue() # integer % complete
81| PM, task = self.PM, self.task
82| lo, hi = PM.getMinimum(), PM.getMaximum()
83| here = int( ( hi - lo ) * 0.01 * progress ) + lo
84| PM.setProgress( here )
85| done = task.isDone()
86| if PM.isCanceled() or done :
87| if not done :
88| task.cancel( 1 )
89| self.button.setEnabled( 1 )
```

Table 18-3 identifies the changes that you nee to make in order to deal with the cancellation of ProgressMonitor or progressTask more appropriately. It is interesting to see how little code is needed in order to adequately handle these cancellation events.[11]

Table 18-3. *Explanation of Changes in ProgressMonitor2.py* **Lines Description**

35-36 The except clause in the progressTask doInBackground() method now silently ignores KeyboardInterrupt exceptions. All others will continue to display information about the exception before the application is terminated.

76 A reference is saved in the object instance to the task object. The PropertyChangeListener event handler needs this reference.
86-88 If the ProgressMonitor was canceled, you might need to cancel the progressTask instance. 89 If the ProgressMonitor was canceled or if the progressTask instance is completed, the Start button will be enabled.
The ProgressMonitor Message

Take another look at the ProgressMonitor constructor in Table 18-2. What data type is the message parameter? It's an object. What does that mean and why isn't it a String? According to the Javadoc, the message argument is an object so that it can be used in different ways, as described in the JOptionPane.message[12] documentation. Additionally, it is important to note that the message portion of the ProgressMonitor object will not change during the life of the ProgressMonitor. Let's see what you can do with the message.

The ProgressMonitor3.py script uses the same technique as shown in Chapter 16 to specify the message as an ImageIcon. I tried this approach first because I wanted to see if I could use this technique to display a different icon on the ProgressMonitor. Unfortunately, as you can see in Figure 18-6, the ImageIcon is displayed in addition to the Information icon.

Figure 18-6. *ProgressMonitor with an ImageIcon message*

Can you use HTML in the message? According to the JOptionPane.message documentation, you would expect strings to be displayed in a JLabel. Since the JLabel class allows HTML text, you might be able to do some interesting things. Figure 18-7 shows what happens when you try using an HTML message, as well as an ordered and unordered list.[13] Unfortunately, it appears that an excessively "long" HTML message can cause some problems. From the little testing that I did, it appears to be related to the horizontal space that is allowed for the message. I guess you need to determine just how much information, and in what form, should be displayed in this message field.

[11] A Java application would need to catch a java.lang.InterruptedException instead.
[12]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JOptionPane.html#message
[13]All of these output images were generated by ProgressMonitor4.py. You will

have to edit the source script and uncomment the desired assignment statement in the start() method to duplicate these different outputs.

Figure 18-7. *ProgressMonitor with HTML message strings*

The ProgressMonitor Note

In the ProgressMonitor constructor in Table 18-2, you can see that, in addition to the message parameter, a note parameter is also defined. What's the difference between the message and note parameters? As you just saw, the message parameter is an object that is represented as a JLabel, so you can use HTML tags to control how it is displayed (within limits). Additionally, it is static for the life of the ProgressMonitor. The note argument is a string that is used to initialize a note attribute. If a value of None (the Jython equivalent to the Java null value) is specified in the constructor, the value of the note attribute shouldn't change.

On the other hand, you can use the note field on the ProgressMonitor to provide information related to the progress that is being made. According to the Javadoc, you only need to call the setNote(...) method to update this portion of the ProgressMonitor. Figure 18-8 shows the output of the ProgressMonitor5.py script, which uses the note field to provide a numeric percentage indicating the progress.

Figure 18-8. *ProgressMonitor with a note attribute from ProgressMonitor5.py*

Does anything about this catch your eye? When I first saw this, I was more than a little surprised by the fact that bottom of the Cancel button was encroaching on the bottom edge of the dialog box. After a bit of testing, I realized what was happening. It appears that the progress monitor's size is based on the initial parameter values. If an empty note string is provided in the ProgressMonitor constructor, then no vertical space is allocated to hold the string when it is changed. How do you fix this? One simple fix is to provide a note string containing one or more blanks. Figure 18-9 shows the output of this same script when the note is initialized with a single blank. I don't know about you, but I think that this looks significantly better.

Figure 18-9. *ProgressMonitor with note initialized with ' '*

Can the note string contain HTML text? Yes it can, but again, I caution you to be careful about the kind of text that you use. It is important to remember that the ProgressMonitor class hierarchy, as shown in Listing 18-11, is based on an object and not a JComponent. So, you don't have the same kind of control over it that you do with normal Swing components, which means that you can't resize it after it is instantiated.

The ProgressMonitor parentComponent

Up to now, you have passed None as the value for the parentComponent argument. What role is this parameter supposed to play? Is there any reason for providing something other than None? The ProgressMonitor6.py script attempts to answer this question. In all of the other ProgressMonitor scripts, None was specified as the parentComponent value. This script references the application by using the self.frame value. Now, regardless of where the frame is positioned on the screen, when you press the Start button, the ProgressMonitor will be positioned with respect to the parentComponent window. This parameter provides the only control over how the ProgressMonitor appears on the screen.

Other ProgressMonitor Properties

One difference that exists between ProgressMonitor objects and other dialog boxes is that there is a delay between when the object is created and when it appears on the screen. Why is that? It's possible for there to be no reason for the ProgressMonitor to be displayed. Certain properties of the ProgressMonitor determine when and if a ProgressMonitor object should be displayed.

If you again take a look at the classInfo function introduced in Chapter 4, as shown in Listing 18-16, you'll see that there are some attributes defined in the ProgressMonitor class that aren't listed in the Javadoc. **Listing 18-16.** ProgressMonitor Attributes

```
wsadmin>from javax.swing import ProgressMonitor wsadmin>
wsadmin>classInfo( ProgressMonitor, attr = '' ) javax.swing.ProgressMonitor

canceled, maximum, millisToDecideToPopup, millisToPopup minimum, note,
progress
| java.lang.Object
* class
| javax.accessibility.Accessible
* accessibleContext
wsadmin>
```

Table 18-4 identifies and describes these attributes. This should help you understand any delay in the appearance of the ProgressMonitor dialog. What happens is that when a progress value is set, the millisToDecideToPopup value determines how long the ProgressMonitor instance waits before trying to determine if the ProgressMonitor should be visible. If the estimated time to

completion is less than the millisToPopup value, the ProgressMonitor dialog will not be displayed. After millisToPopup milliseconds, the dialog will appear. Once the progress value reaches the maximum value, the ProgressMonitor is hidden.

Table 18-4. *ProgressMonitor Attributes, Explained*

Attribute Name	Description
canceled	Boolean value indicating if the Cancel button has been used.
maximum	Integer value initialized by constructor; accessible via getter and setter methods.
millisToDecideToPopup	Integer value defaulting to 500 (0.5 sec); accessible via getter and setter methods.
millisToPopup	Integer value defaulting to 2000 (2 sec); accessible via getter and setter methods.
minimum	Integer value initialized by constructor; accessible via getter and setter methods.
note	String value initialized by constructor; accessible via getter and setter methods. **Note:** If this is initialized to None, any changes made via setNote() are ignored.
progress	Write-only integer value modified via the setProgress() setter method. Values are limited by the values of minimum and maximum. **Note:** When a progress is set to the maximum, the ProgressMonitor is hidden.

One question that keeps coming up with respect to ProgressMonitor dialog boxes relates to the fact that the dialog box is hidden when the process value is set to the maximum value. Generally, developers wonder whether there is any way to get the dialog box to stay visible once the maximum value is set. Unfortunately not. However, you can pass the setProgress(...) method a value in the range from minimum to maximum - 1. Unfortunately, if the range is small (e.g., 0 .. 10), a value of maximum - 1 might leave a visible gap at the upper end of the progress bar. In this case, you might want to consider using some kind of multiplier so that the value of maximum - 1 won’t be discernable.

Another possible annoyance related to doing this exists. Once the dialog is hidden, if you use the setProgress(...) method to cause it to reappear, it will not

be located where the user left it. It will be located based on the original position, relative to the value of the parentComponent attribute. So, if the user moves the ProgressMonitor dialog box, the dialog box will appear to jump from where it is back to its original screen location.

ProgressMonitorInputStream Objects

There is another class, much like ProgressMonitor, that can use a ProgressMonitor object to determine the progress while data is read from an InputStream. As with ProgressMonitor objects, the ProgressMonitorInputStream[14] processing (reading data from the InputStream) should be performed by a separate (SwingWorker) type of task.

Unfortunately, space and time limitations don't allow me to provide more detail about this particular class.

Summary

This chapter covered monitoring and communicating progress to the user with a variety of techniques. The really nice thing that this shows you is how much control application developers have over the way they convey progress to their users. It's important to understand the implications and consequence of each choice you make, so keep this in mind. In next chapter, you'll learn about internal frames, which allow you to create even more interesting applications.

[14]See http://docs.oracle.com/javase/8/docs/api/javax/swing/ProgressMonitorInputStrea

Chapter 19

Internal Frames

Up to this point, the applications you've seen have been able to create and use individual windows. Almost every sample has used a single JFrame to display information and interact with the users. Now you're going to look at using internal frames. First you will take a quick look at a collection of inner frames, because there are times when applications can use a variety of views in order to convey different information to the users. The chapter begins by comparing internal frames to the JFrame used in the previous applications. Then you'll

build an application that uses internal frames that display information in a variety of ways.

Looking at Inner Frames

Figure 19-1 shows the output of a simple application that displays three inner frames.

Figure 19-1. *Simple application with three inner frames*

Listing 19-1 shows the code required to produce this kind of simple inner frame. As you can see, even though very little code is required, it is quite functional.[1] The script, which you can find in the code\Chap_19\iFrameDemo.py file, shows that each of the inner frames can be moved, resized, closed, iconified, restored, and even maximized within the available space.

Listing 19-1. iFrameDemo Class Within iFrameDemo.py

8|class iFrameDemo(java.lang.Runnable) :
9| def run(self) :
10| screenSize = Toolkit.getDefaultToolkit().getScreenSize() 11| w = screenSize.width >> 1 # 1/2 screen width 12| h = screenSize.height >> 1 # 1/2

screen height 13| x = (screenSize.width - w) >> 1
14| y = (screenSize.height - h) >> 1
15| frame = JFrame(
16| 'iFrameDemo',
17| bounds = (x, y, w, h), # location & size 18| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 19|)
20| desktop = JDesktopPane()
21| for i in range(3) :
22| inner = JInternalFrame(
23| 'Inner Frame #%d' % (i + 1),
24| 1, # Resizeable 25| 1, # Closeable 26| 1, # Maximizable 27| 1, # Iconifiable
28| visible = 1, # setVisible(1) 29| bounds = (i * 25 + 25, i * 25 + 25, 250, 250) 30|)
31| desktop.add(inner)
32| frame.setContentPane(desktop)
33| frame.setVisible(1)

Table 19-1 describes each of the steps in Listing 19-1. You will be learning about these steps in more detail throughout this chapter.

[1]Please note, however, that the iFrameDemo.py script output requires user manipulation to look like the image seen in Figure 19-1. This is discussed further in section 19-2.

Table 19-1. *iFrameDemo Class, Explained*

Lines Description

10 Determines the size of the current screen.
11 Makes the width of the application use half the physical width of the screen.
12 Makes the height of the application use half the physical height of the screen.
13-14 Computes the upper-left corner of the application to center the application window. 15-19 Instantiates the frame using the specified parameters.
20 Creates a JDesktopPane onto which the InternalFrames will be added.

21-31 Creates three InternalFrames, one at a time, and adds them to the desktop. **Note:** The bounds keyword argument is used to size and position the inner frame on the JDesktopPane. 32 Replaces the frame ContentPane with the populated desktop.
33 Makes the application frame visible.

Before you investigate the JDesktopPane[2] class shown in Listing 19-1, it's a good idea to learn more about layers and about the JLayeredPane[3] on which this

class is based.

Layers

Let's take a quick look at a simple script that demonstrates layered components using a trivial component—the JLabel object. Figure 19-2 shows a group of seven overlapping colored labels. Note how the text is centered across the top of each label, and how the labels are stacked to make all of the text visible.

Figure 19-2. *Layered labels*

[2]http://docs.oracle.com/javase/8/docs/api/javax/swing/JDesktopPane.html.

[3]http://docs.oracle.com/javase/8/docs/api/javax/swing/JLayeredPane.html.

Listing 19-2 shows the relevant part of the LayeredPaneDemo class from the LayeredPaneDemo.py script.

Listing 19-2. LayeredPaneDemo Class, Part 1

```
11|class LayeredPaneDemo( java.lang.Runnable ) :
12| def run( self ) :
| ...
18| frame.setContentPane( self.createLayeredPane() )
19| frame.pack()
20| frame.setVisible( 1 )
21| def createColoredLabel( self, text, color ) :
```

```
22| return JLabel(
23| text,
24| opaque = 1,
25| size = ( 150, 130 ),
26| background = color,
27| foreground = Color.black,
28| verticalAlignment = JLabel.TOP,
29| horizontalAlignment = JLabel.CENTER,
30| border = BorderFactory.createLineBorder( Color.black )
31| )
```

Unfortunately, this class is too long to fit onto one page, so it has to be split into pieces. Table 19-2 describes the portion of the LayeredPaneDemo shown in Listing 19-2.

Table 19-2. *Comments About the LayeredPaneDemo Class, Part 1*

Lines	Description
12-20	This portion of the run() method should be familiar to you by now. The only new part is where the default ContentPane is replaced by the result of calling the createLayeredPane() in line 18.
21-31	The createColoredLabel(...) method creates and returns a JLabel containing the specified text and using the specified color. Each label object is roughly square, and the text is positioned in the center of the top, with a black border.

The remainder of the LayeredPaneDemo class, the createLayeredPane(...) method, is shown in Listing 19-3. The purpose of this method is to create and return a layered pane object populated with a collection of colored labels.

Listing 19-3. LayeredPaneDemo Class, Part 2

```
32| def createLayeredPane( self ) :
33| colors = [
34| ( 'Red' , Color.red ),
35| ( 'Orange', Color.orange ),
36| ( 'Yellow', Color.yellow ),
37| ( 'Green' , Color.green ),
38| ( 'Blue' , Color.blue ),
```

```
39| ( 'Indigo', Color( 75, 0, 130 ) ),
40| ( 'Violet', Color( 143, 0, 255 ) )
41| ]
42| result = JLayeredPane(
43| border = BorderFactory.createTitledBorder( 44| 'Layered Pane'
45| ),
46| preferredSize = Dimension( 290, 280 ) 47| )
48| position, level = Point( 10, 20 ), 0 49| for name, color in colors :
50| label = self.createColoredLabel( 51| 'Layer %d = %s' % ( level, name ), 52|
color
53| )
54| label.setLocation( position )
55| position.x += 20
56| position.y += 20
57| result.add( label, level, 0 )
58| level += 1
59| return result
```

Table 19-3 describes the statements found in this method.

Table 19-3. *Comments About the LayeredPaneDemo Class, Part 2*

Lines Description

32-59 The createLayeredPane(...) method is used to create, populate, and return a JLayeredPane object instance.

33-41 The colors array holds information about the text and colors to be used to create each of the label objects.

42-47 Creates the JLayeredPane object with the specified preferred size and a title.

48 Each label will be positioned in an overlaid fashion, at a different layer on the pane, starting with these values.

49-58 Loops over the available colors (defined in 33-41), creating and positioning the labels accordingly. 57 This statement adds the current label to the layered pane at the specified layer and proper position. I will discuss this method in more detail shortly.

58 Increments the level variable used to indicate the layer number of the next label.

59 Returns the populated JLayeredPane object instance.
Position Within the Layer

You might wonder what the third parameter on the add(...) method (line 57 in Listing 19-3) is used for. It identifies the position, within the layer, of the component being added. Figure 19-3 shows what happens when you don't include the position parameter. Notice how in Figure 19-2, the top portion of each label (including the text) was visible.

Figure 19-3. *Layered Labels without the position parameter specified*

As you can see, the layers are now stacked in reverse order, with the red label on top. If you add a print statement after the add() method statement and display the value of level (the intended level of the component being added) and the result of calling the getLayer(...) method for the label component on the layered panel, you'll see that an unintended add(...) method is being invoked and you aren't specifying the level, as expected. Listing 19-4 shows the proposed statement changes.

Listing 19-4. Changes to the createLayeredPane() Method

```
| ...
57| result.add( label, level, 0 )
```

```
58| print level, result.getLayer( label )
| ...
```

The output of the getLayer(...) method call for each label will show that every label is, in fact, being added to layer zero. With the three argument add(...) method calls, the output of the getLayer(...) method call will show the expected layer value.

So, what is the position supposed to do and when does it come into play? It is used when multiple components are at the same layer. It is used to determine the relationship of components on the same level. The position value should be an integer from -1 to N - 1, where N is the number of components in layer. The larger the value (the closer to N) of the position, the deeper the component.

The exception to this is -1, which is considered the same as N - 1, which means that it will be the deepest component on the layer. That's why you specify a value of zero on the add(...) method when you add a component to a layer. This will position the newest component as the uppermost position within the layer (closest to the user).

The JDesktopPane Class

Now that you've seen how layers work, you'll be better able to understand the JDesktopPane class. It is generally best to have internal frames added to a JDesktopPane instance in order to show and manipulate them. If you look back at Figure 19-1, you should notice familiar JFrame icons manipulating the internal frames. It should be clear how different these are from something as simple as a label.

There was another reason for discussing the JLayeredPane before moving onto the JDesktopPane, and that is the add(...) method as shown in Listing 19-1, line 31. If you take another look at Figure 19-1, you'll see that the internal frames have been moved around so that they don't overlap. This is because the initial output of the iFrameDemo script has the layering in an unexpected order. Unexpected until you understand which of the add(...) methods should have been used. The single argument add(...) method is used in Listing 19-1. If, on the other hand, you had used the three argument version, as shown in line 57 of Listing 19-3, the first internal frame (Inner Frame #3) would have been completely visible. Figure 19-4 shows the ordering of the internal frames when

using the single and multiple argument add(...) methods.[4] As a user of GUI applications, which output would you prefer?

Figure 19-4. *iFrameDemo using single (top) and multiple (bottom) argument* add(...) *methods*

[4] The three argument add(...) method call can be seen as a comment in the iFrameDemo.py sample script. The Javadoc for which is http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html#add%28java.av %20java.lang.Object,%20int%29.

Another thing to note about these sample scripts is that when the inner frames are created, their position needs to be specified so that all of the inner frames aren't stacked on top of each other, thereby hiding the frames underneath.

One thing that you haven't learned about in much detail yet is the JDesktopPane instance onto which all of these inner frames are added. In fact, I have only mentioned its add(...) method. There have to be more, don't there? Of course there are more. In fact, there are far too many to discuss in this chapter. In fact, the sample scripts in this chapter don't use anything but the add(...) method. That doesn't mean that they aren't useful. I just don't have the space or time required to do each of them justice.

More complex scripts are likely to use the JDesktopPane methods to manipulate the inner frames. To give you a little idea what some of the methods can do, I've listed a few of the most useful in Table 19-4. Remember that this is only a partial list.

Table 19-4. *Some Useful JDesktopPane Methods*
Returned Value
JInternalFrame[]

Method Signature and Role
getAllFrames()
Returns all JInternalFrames currently present in the desktop container.

JInternalFrame[] getAllFramesInLayer(int layer)
Returns all JInternalFrames currently contained at the specified layer of the desktop object.

JInternalFrame getSelectedFrame()
Returns the currently active JInternalFrame in the desktop, or None if none is selected. void remove(int index)
Removes the specified component from the container.
void removeAll()

Removes all the components from the container.

JinternalFrame selectFrame(boolean forward)
Selects the next component in the container if the value of forward is true (1) or the previous component if the forward value is false (0).

void setSelectedFrame(JInternalFrame f)
Makes the specified JInternalFrame active.

JFrame or JInternalFrame?

When trying to decide how to implement an application, the similarity of the JFrame and JInternalFrame[5] classes might lead you to think that you can start with one and easily switch to the other should the need arise. Unfortunately, it's not quite as simple as that. It's better to pick one from the start and stick with it. It's possible that you could pick one way and soon realize that the other technique would have been better. If you haven't invested too much time or effort, it might be better to start over at that point. Here are some questions to consider to help you decide which approach is more advantageous for your needs.

• How contained and well defined is your application? Does it make more sense for you to see one, or multiple kinds or views of data at the same time? If you only need to see one view, then a single frame, possibly with something like tabbed panes might make more sense.

• How much data sharing will you need to do for the different portions of your application? If each aspect of your application is separate and distinct, there might be some advantage to seeing multiple views or pieces of data. If this is the case, multiple internal frames may be a better approach.

• Do multiple instances of the same kind of view make better sense, or is a single perspective sufficient? Might multiple views add complexity or confusion? If multiple, simultaneous views are advantageous, using multiple internal frames might be the best approach. However, try to consider the additional complexity that might be required by this choice.

One of the really powerful things about using Jython to produce or prototype your graphical application is how easy it is to create a proof of concept. This

allows you to start your development and improve your knowledge about what exactly you want your program to do. This might help you decide which approach is best—a single frame or multiple internal frames.

The JInternalFrame Class

One way to determine the difference between the JInternalFrame and JFrame classes is by comparing the methods available to each. A quick glance shows that the JFrame class has about 30 methods and the JInternalFrame class has about 80. This would appear to be a huge difference, which in some ways, it is. Looking a bit closer at the methods, you'll see that the JInternalFrame class has a large number of methods (46) that are identified as protected, but the JFrame class has only eight. Additionally, the JInternalFrame class has some deprecated methods.

JFrame and JInternalFrame Methods

While on the topic of graphical applications, consider what it would take to create a small application that showed the JFrame and JInternalFrame methods side-by-side and determined whether the protected and deprecated methods should be viewable. Figure 19-5 shows an image from this simple application. The menu selections allow you to hide the deprecated and protected methods.

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JInternalFrame.html.

Figure 19-5. *FrameMethod application images*

To create this fairly simple application, you first have to figure out where the data exists and in what format. Based on this, you need to figure out how to get the data into your application. For simplicity's sake, I created two text files, one for each class, and one line for each method. Each line has three fields:

- The optional modifier and return type of the method
- The method signature (the method name and parameter list)
- The method abstract

The fields are separated by the | delimiter. So, the application needs to load the text from the file and process each line. Listing 19-5 contains some utility methods from the FrameMethods.py script.[6] The textFile(...) method returns a string containing the file contents and the parse(...) method formats each line to contain only the first two fields from the file, with all of the method names aligned on the first character in the method name.

Listing 19-5. FrameMethods.py Utility Methods

```
25| def parse( self, text ) :
26| data = [ line.split( ' | ' ) for line in text.splitlines() ]
27| width = max(
28| [ len( result ) for result, sign, desc in data ]
29| )
30| return '\n'.join(
31| [
32| '%*s %s' % ( width, result, sign ) 33| for result, sign, desc in data 34| ]
35| )
| ...
111| def textFile( self, filename ) :
112| result = ''
113| try :
114| f = open( filename )
115| result = f.read()
116| f.close()
117| except :
118| Type, value = sys.exc_info()[ :2 ] 119| result = '%s\n%s' % ( Type, value )
120| return result
```

[6]See ...\code\Chap_19\FrameMethods.py in the sample directories.

Listings 19-6 and 19-17 show the run(...) method from the FrameMethods class, which is used to create the application frame, define the layout, load the files, and populate the panes and the application menu. The contents and structure of this method should be quite familiar to you by now, so I won't bother describing it in detail.

Listing 19-6 shows the run(...) method of the FrameMethods.py script. In it, you can see how the frame is composed of two JTextArea sections (lines 47-53 and lines 58-64), each of which is contained in a JScrollPane (lines 54 and 65). Each text area is populated from one of the text files used by this script.

Listing 19-6. FrameMethods run() Method

```
15|class FrameMethods( java.lang.Runnable ) :
| ...
36| def run( self ) :
37| frame = JFrame(
38| 'FrameMethods',
39| size = ( 1000, 500 ),
40| locationRelativeTo = None,
41| layout = GridLayout( 0, 2 ),
42| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
43| )
44| self.one = self.parse(
45| self.textFile( 'JFrame Methods.txt' )
46| )
47| self.left = JTextArea(
48| self.one,
49| 20,
50| 40,
51| editable = 0,
52| font = Font( 'Courier' , Font.PLAIN, 12 )
53| )
54| frame.add( JScrollPane( self.left ) ) 55| self.two = self.parse(
56| self.textFile( 'JInternalFrame Methods.txt' ) 57| )
58| self.right = JTextArea(
59| self.two,
60| 20,
61| 40,
```

```
62| editable = 0,
63| font = Font( 'Courier' , Font.PLAIN, 12 ) 64| )
65| frame.add( JScrollPane( self.right ) ) 66| frame.setJMenuBar(
self.makeMenu() )
67| frame.setVisible( 1 )
```

In case you are interested, the instance variables one and two are used by the event handler methods to determine the text that will be displayed in each text area. Listing 19-7 shows the makeMenu() method from the same script. As you can see, this method creates, populates, and returns the menu bar and its entries, which have the associated actionPerformed event handling methods assigned as the showItems() method.

Listing 19-7. FrameMethods makeMenu() Method

```
68| def makeMenu( self ) :
69| menuBar = JMenuBar(
70| background = Color.blue,
71| foreground = Color.white
72| )
73| showMenu = JMenu(
74| 'Show',
75| background = Color.blue,
76| foreground = Color.white
77| )
78| self.deprecated = JCheckBoxMenuItem(
79| 'Deprecated',
80| 1,
81| actionPerformed = self.showItems
82| )
83| showMenu.add( self.deprecated )
84| self.protected = JCheckBoxMenuItem(
85| 'Protected',
86| 1,
87| actionPerformed = self.showItems
88| )
89| showMenu.add( self.protected )
90| showMenu.addSeparator()
91| showMenu.add(
```

```
92| JMenuItem(
93| 'Exit',
94| actionPerformed = self.exit
95| )
96| )
97| menuBar.add( showMenu )
98| return menuBar
```

Listing 19-8 shows the showItems(...) method used as the menu item ActionListener event handler. It, in turn, may use the findNot(...) method to filter the data, removing all lines containing the specified text (either “deprecated” or “protected”).

Listing 19-8. The showItems() and findNot() Methods

```
18| def findNot( self, data, text ) :
19| return '\n'.join(
20| [
21| line for line in data.splitlines()
22| if line.lower().find( text ) < 0
23| ]
24| )
| ...
99| def showItems( self, event ) :
100| item = event.getActionCommand()
101| one = self.one
102| two = self.two
103| if not self.deprecated.isSelected() : 104| one = self.findNot( one,
'deprecated' ) 105| two = self.findNot( two, 'deprecated' ) 106| if not
self.protected.isSelected() : 107| one = self.findNot( one, 'protected' ) 108| two =
self.findNot( two, 'protected' ) 109| self.left.setText( one )
110| self.right.setText( two )
```

I’ll admit that I created this application quickly. It was one of those “rapid prototyping” opportunities that took about two hours, all told. Most of that time was spent creating the input files. It only required about 30 minutes to create the actual application, once I had the data in place. Not bad for a quick “proof of concept,” wouldn’t you say? The really neat part was that it is more than adequate, as is, to deal with the menu selection events. At first, I wasn’t sure how rapidly the application would update after a menu selection was made.

Watching it in action, though, I was pleasantly surprised and am very happy with the responsiveness.

Does that mean that this is the best way to go? Absolutely not. That's one of the great things about being able to create a quick proof of concept script like this. It can help you understand the problem better, as well as give you some ideas about improvements and additional features. For example, based on this example, I can imagine a more useful application that:

• Adds menu items to identify the classes to be compared, such as allows users to specify input sources (URL or filename).

• Uses jsoup (Chapter 15) to retrieve and process the Javadoc URLs.
• Uses the JList class (Chapter 9) to allow a method to be selected. If the same method signature exists in the other list, highlights it as well.

• Adds selection logic to highlight the corresponding method in the other pane, if one exists.

In fact, the more I think about it, this might be a really good enhancement to the javadocInfo script in Chapter 15. What do you think? However, since we're discussing internal frames maybe that would be a good way to do it. . . It's a thought.

More JFrame and JInternalFrame Differences

One reason for looking at the differences between the JFrame and JInternalFrame classes is to understand the extent. Besides looking at their methods, you might also wonder about the differences in their class hierarchies. Listing 19-9 shows the output of the classInfo function, which was introduced in Chapter 4. It emphasizes that even though the class names are similar, they are very different. One of the most important differences was pointed out in Chapter 1, where the JFrame class was identified as a top-level container.

Listing 19-9. JFrame and JInternalFrame Class Hierarchies

```
wsadmin>from javax.swing import JFrame, JInternalFrame wsadmin>
wsadmin>classInfo( JFrame )
javax.swing.JFrame
| java.awt.Frame
```

```
| | java.awt.Window
| | | java.awt.Container
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver
| | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | javax.accessibility.Accessible
| | java.awt.MenuContainer
| javax.swing.WindowConstants
| javax.accessibility.Accessible
| javax.swing.RootPaneContainer
wsadmin>
wsadmin>classInfo( JInternalFrame )
javax.swing.JInternalFrame
| javax.swing.JComponent
| | java.awt.Container
| | | java.awt.Component
| | | | java.lang.Object
| | | | java.awt.image.ImageObserver
| | | | java.awt.MenuContainer
| | | | java.io.Serializable
| | java.io.Serializable
| javax.accessibility.Accessible
| javax.swing.WindowConstants
| javax.swing.RootPaneContainer
wsadmin>
```

The differences between these two classes are significant, even though the function they provide is quite similar in nature. One area that differs significantly is the type of events recognized by each class.

JInternalFrame Events

In order to work well with internal fames, you need to understand the events that can occur in order to decide how your application should react when these events occur. The Javadoc for the JInternalFrame class identifies the InternalFrameListener[7] class as the base class for dealing with these kinds of events. Since it is an identified as an interface, it is probably better idea to consider using the InternalFrameAdapter[8] class as a base class.

What would it take to create an application that uses internal frames to help you better understand internal frame events? You can use one perpetual internal frame to display details about the internal frame events when they occur.[9] All of the others can be simple internal frames that can be created, minimized, restored, maximized, and closed.

The perpetual inner frame should contain a scrollable text area. This would allow your application to display the details and information about the events as they occur. Taking a quick look back to Listing 19-1, you can make some decisions about what arguments should be used to create this special internal frame. It would probably make the application much more complex if you allowed zero or multiple instances of this special internal frame to exist. So let's start simply by allowing only one perpetual instance in your application.

Listing 19-10 shows the first attempt of implementing a logging internal frame class. As you can see, it's really simple and straightforward. In fact, there really shouldn't be anything surprising about it.

Listing 19-10. The eventLogger Class, First Attempt from iFrameEvents1.py

```
10|class eventLogger( JInternalFrame ) :
11| def __init__( self ) :
12| JInternalFrame.__init__(
13| self,
14| 'eventLogger',
15| 1, # Resizeable - yes
16| 0, # Closeable - no
17| 0, # Maximizable - no
18| 1, # Iconifiable - yes
19| visible = 1,
20| bounds = ( 0, 0, 250, 250 )
21| )
22| self.textArea = JTextArea(
23| 20, # rows
24| 40, # columns
25| editable = 0 # read-only
26| )
27| self.add( JScrollPane( self.textArea ) )
```

When instantiated, this class creates an internal frame, positions it, defines its size, and adds a scrollable pane containing a read-only text area to it. Wow, that

was easy. Figure 19-6 shows the top-left portion of an application showing this internal frame.

[7] See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/InternalFrameListene
[8]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/InternalFrameAdapte
[9]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/InternalFrameEvent.h

Figure 19-6. *iFrameEvents1.py initial display*

That's nice, but how is it supposed to log an InternalFrameEvent? How is the application supposed to create any other internal frames?
The first issue is easy enough to solve. You can have the eventLogger class instantiate an InternalFrameAdapter instance and provide a getter method to access it. This means that when any InternalFrameEvent occurs, the event handler will be able to access the event logger frame and add text to it.
The second issue is solved by adding a simple menu that allows you to create more internal frames.
Most of the revised script is in code\Chap_19\iFrameEvents2.py and is shown in the following listings. Listing 19-11 shows the utility methods for this class, which should look quite familiar by now.

Listing 19-11. Beginning of the iFrameEvents Class from iFrameEvents2.py

```
56|class iFrameEvents2( java.lang.Runnable ) :
57| def addIframe( self, event ) :
58| desktop = self.desktop
59| self.iFrameCount += 1
60| i = self.iFrameCount % 10
61| inner = JInternalFrame(
62| 'Inner Frame #%d' % self.iFrameCount,
63| 1, # Resizeable
64| 1, # Closeable
65| 1, # Maximizable
66| 1, # Iconifiable
67| bounds = ( i * 20 + 20, i * 20 + 20, 200, 200 )
68| )
69| inner.addInternalFrameListener( self.logger.getListener() )
70| inner.setVisible( 1 )
71| desktop.add( inner, i, 0 )
72| def exit( self, event ) :
73| sys.exit()
74| def menuBar( self ) :
75| result = JMenuBar()
76| newMenu = result.add( JMenu( 'New' ) ) 77| newMenu.add(
78| JMenuItem(
79| 'InnerFrame',
80| actionPerformed = self.addIframe 81| )
82| )
83| newMenu.addSeparator()
84| newMenu.add(
85| JMenuItem(
86| 'Exit',
87| actionPerformed = self.exit 88| )
89| )
90| return result
```

The addIframe(...) method, shown in lines 57-71, is called by the menu entry event handler to create a new internal frame. It is important to note that this method depends on the self.logger instance attribute value (see line 69), which is initialized in the run() method.

Listing 19-12 shows the run() method, which:
• Creates the initial application frame of the initial size and location
• Creates and adds the menu bar
• Creates a desktop instance to hold the internal frames
• Creates the special internal frame, which is an instance of the eventLogger class
• Adds the special eventLogger instance to the desktop
• Replaces the initial frame ContentPane with the desktop
• Makes the application visible

Listing 19-12. The run() Method from the iFrameEvents Class from iFrameEvents2.py

```
91| def run( self ) :
92| screenSize = Toolkit.getDefaultToolkit().getScreenSize()
93| w = screenSize.width >> 1 # 1/2 screen width
94| h = screenSize.height >> 1 # 1/2 screen height
95| x = ( screenSize.width - w ) >> 1
96| y = ( screenSize.height - h ) >> 1
97| frame = JFrame(
98| 'iFrameEvents2',
99| bounds = ( x, y, w, h ), # location & size 100| defaultCloseOperation =
JFrame.EXIT_ON_CLOSE 101| )
102| frame.setJMenuBar( self.menuBar() )
103| self.desktop = desktop = JDesktopPane()
104| self.logger = eventLogger()
105| desktop.add( self.logger, 0, 0 )
106| frame.setContentPane( desktop )
107| self.iFrameCount = 0
108| frame.setVisible( 1 )
```

The event listener that is added to each internal frame was instantiated by the run() method and is an instance of the eventLogger class, as shown in Listing 19-13. In fact, I added a getter for the eventAdapter instance, as shown in lines 54 and 55.

Listing 19-13. The eventLogger Class from iFrameEvents2.py

```
34|class eventLogger( JInternalFrame ) :
35| def __init__( self ) :
```

```
36| JInternalFrame.__init__(
37| self,
38| 'eventLogger',
39| 1, # Resizeable - yes
40| 0, # Closeable - no
41| 0, # Maximizable - no
42| 1, # Iconifiable - yes
43| visible = 1,
44| bounds = ( 0, 0, 250, 250 )
45| )
46| self.textArea = JTextArea(
47| '', # Inital text
48| 20, # rows
49| 40, # columns
50| editable = 0 # read-only
51| )
52| self.eventListener = eventAdapter( self.textArea )
53| self.add( JScrollPane( self.textArea ) )
54| def getListener( self ) :
55| return self.eventListener
```

Listing 19-14 shows the eventAdapter class that is used by the eventLogger class. It is important to note how the constructor, in lines 16-18, requires the text area to be updated by the messages that this class needs to log. The log(...) utility method, found on lines 19-26, uses a regular expression (RegExp) to extract the name for the kind of event being logged and updates the specified text area with a line identifying the title of the internal frame for which the event occurred, as well as the kind of event.

■**Note** notice how all of the event handler methods are trivial calls to the log() method, which then does all of the real work.

Listing 19-14. The eventAdapter Class from iFrameEvents2.py

```
15|class eventAdapter( event.InternalFrameAdapter ) :
16| def __init__( self, textArea ) :
17| self.textArea = textArea
18| self.regexp = re.compile( '\[INTERNAL_FRAME_(\w+)]' )
19| def log( self, event ) :
20| title = event.getInternalFrame().getTitle()
```

```
21| mo = re.search( self.regexp, event.toString() )
22| if mo :
23| Type = mo.group( 1 ).capitalize()
24| else :
25| Type = 'unknown'
26| self.textArea.append( '\n%s : %s' % ( title, Type ) ) 27| def
internalFrameActivated( self, ife ) : self.log( ife ) 28| def internalFrameClosed(
self, ife ) : self.log( ife ) 29| def internalFrameClosing( self, ife ) : self.log( ife )
30| def internalFrameDeactivated( self, ife ) : self.log( ife ) 31| def
internalFrameDeiconified( self, ife ) : self.log( ife ) 32| def
internalFrameIconified( self, ife ) : self.log( ife ) 33| def internalFrameOpened(
self, ife ) : self.log( ife )
```

I'll let you play with the sample application to get a feel for the kinds of events that are generated and the order in which they can occur.

More JInternalFrame Topics

Unfortunately, internal frames, and the manipulation thereof, are not simple or easy. There are lots of things to deal with. For example, you can use the simple iFrameEvents2.py script to get a better idea.

1. Start the script.
2. Choose New ➤ InnerFrame to create Inner Frame #1.
3. Use the minimize icon on Inner Frame #1 to iconify it.
4. Resize the application to hide the iconified inner frame.

How are you supposed to access the iconified inner frame without making the application larger? This is a simple example of one of the many kinds of things that you need to consider when you are dealing with inner frames. It may also help you understand the need for some of the JDesktopPane methods listed in Table 19-4.

Building an Application from Scratch

This section shows what it takes to build an application that uses the JInternalFrame class. Rather than showing the end result, the section iterates with fairly simple steps to create the script from absolutely nothing, to the point where you'll have a better understanding of what it takes to put these Swing components together into some semblance of a complete application.

Where should you start? Let's look for something, hopefully simple, that you can use to make an interesting application. From the WebSphere Application Server online documentation, you can find a simple script written in Jacl that can be used to change the console timeout value.[10] This value determines the length of time (in minutes) that the administrator console will remain active without user input.

Since the script in the documentation is written in Jacl, you'll begin by taking a look at it. In the code directory for this chapter, you can find the consoleTimeout_00.jacl script (see code\Chap_19\consoleTimeout_00.jacl). It is slightly different from the online documentation. It is different in that it allows users to specify the new timeout value on the wsadmin command line. The script in the documentation requires you to edit the script file and replace the two instances of <timeout value> with the desired timeout value (in minutes).

[10]See http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.nd.doc/info/ ae/isc/cons_sessionto.html.

Figure 19-7 shows an example invocation of this wsadmin script, which presumes that the current working directory contains the WebSphere Application Server wsadmin.sh script.

```
./wsadmin.sh -f consoleTimeout.jacl 30
```

Figure 19-7. *Using the consoleTimeout_00.jacl script*

Simple Non-GUI Jython Version of the consoleTimeout Script

I don't know about you, but I find it a challenge to write and modify Jacl scripts.[11] So, let's start by converting this Jacl script to a roughly equivalent Jython script. Listings 19-14 and 19-15 show the consoleTimeout routine from this converted script. During the conversion, a little usability enhancement was added to display the current timeout value if no command-line parameters are provided. Additionally, if too many parameters are specified, some usage information is displayed.

Listing 19-15. The consoleTimeout Routine from consoleTimeout_01.py, Part 1

```
23|def consoleTimeout_01( cmdName = 'consoleTimeout_01' ) :
24| argc = len( sys.argv ) # Number of args
25| if argc > 1 : # Too many?
26| Usage( cmdName ) # show Usage info
```

```
27| value = None
28| if argc == 1 :
29| value = sys.argv[ 0 ]
30| if not re.search( re.compile( '^\d+$' ), value ) :
31| print nonNumeric % locals()
32| Usage( cmdName )
33| dep = AdminConfig.getid( '/Deployment:isclite/' )
34| if not dep :
35| print noISCLite % locals()
36| Usage( cmdName )
37| appDep = AdminConfig.list( 'ApplicationDeployment', dep )
38| appConfig = AdminConfig.list( 'ApplicationConfig', appDep )
39| if not appConfig :
40| appConfig = AdminConfig.create(
41| 'ApplicationConfig',
42| appDep,
43| []
44| )
45| sesMgmt = AdminConfig.list( 'SessionManager', appDep )
46| if not sesMgmt :
47| sesMgmt = AdminConfig.create(
48| 'SessionManager',
49| appConfig,
50| []
51| )
```

This script should be pretty straightforward to those with a modicum of wsadmin scripting experience. It verifies that the user-supplied valid numeric input then uses some calls to the wsadmin scripting objects to locate the session manager configuration object so that its tuning parameters can be checked. Listing 19-16 deals with creating tuning parameters in order to set the invalidationTimeout attribute value. One interesting thing about this simple script is the fact that almost 60 lines are needed to provide this simple functionality.

[11]I'll admit it, it took me much longer than it should have to figure out how to modify the consoleTimeout.jacl script to accept and use command-line parameters. I am not proficient with Jacl since I use it so infrequently.

■ **Note** In case you are interested, you can look at the complete source code to see that the lines that aren't shown in these listings are comments, initialization

statements, and a Usage(...) method, which displays appropriate information about how the script can and should be used.

Listing 19-16. The consoleTimeout Routine from consoleTimeout_01.py, Part 2

```
52| tuningParams = AdminConfig.showAttribute(
53| sesMgmt,
54| 'tuningParams'
55| )
56| if value :
57| if not tuningParams :
58| AdminConfig.create(
59| 'TuningParams',
60| sesMgmt,
61| [[ 'invalidationTimeout', value ]]
62| )
63| else :
64| AdminConfig.modify(
65| tuningParams,
66| [[ 'invalidationTimeout', value ]]
67| )
68| else :
69| if not tuningParams :
70| print noTPobj % locals()
71| else :
72| timeout = AdminConfig.showAttribute(
73| tuningParams,
74| 'invalidationTimeout'
75| )
76| print currentVal % locals()
77| if AdminConfig.hasChanges() :
78| print saveConfig % locals()
79| AdminConfig.save()
```

First GUI Jython Version of the consoleTimeout Script

The first Jython Swing version of this script is shown in Listings 19-16 and 19-17. Remember that first impressions can be deceiving. Even though there appear to be many more statements in the Swing version (consoleTimeout_02.py), that’s

simply because of the number of statements displayed on multiple lines so that they fit in the space available in the listings in this publication.

Listing 19-17 contains most of the run(...) method from the ConsoleTimeout class in this script. Note that very little effort was put into any complex layout of the Swing components. If you make the frame wider, I think that you'll agree that it doesn't look as nice as it could.

Listing 19-17. The ConsoleTimeout Class run(...) Method from consoleTimeout_02.py

```
9|class consoleTimeout_02( java.lang.Runnable ) :
10| def run( self ) :
11| frame = JFrame(
12| 'consoleTimeout_02',
13| layout = FlowLayout(),
14| size = ( 180, 120 ),
15| locationRelativeTo = None,
16| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 17| )
18| dep = AdminConfig.getid( '/Deployment:isclite/' )

| ...
48| if not self.tuningParms :
49| timeout = ''
50| messageText = "tuningParams object doesn't exist."
51| else :
52| timeout = AdminConfig.showAttribute(
53| self.tuningParms,
54| 'invalidationTimeout'
55| )
56| messageText = ''
57| frame.add( JLabel( 'Timeout:' ) )
58| self.text = frame.add(
59| JTextField(
60| 3,
61| text = timeout,
62| actionPerformed = self.update
63| )
64| )
65| frame.add( JLabel( 'minutes' ) )
```

```
66| self.message = frame.add( JLabel( messageText ) ) | ...
76| frame.setVisible( 1 )
```

Listing 19-18 contains the update(...) method from the same ConsoleTimeout class, which is almost identical to the previous code. It is important to remember that this method is an event handler.

Listing 19-18. The ConsoleTimeout Class update(...) Method from consoleTimeout_02.py

```
77| def update( self, event ) :
78| value = self.text.getText()
79| if not re.search( re.compile( '^\d+$' ), value ) :
80| text = 'Invalid numeric value: "%s"' % value
81| else :
82| if not self.tuningParms :
83| try :
84| AdminConfig.create(
85| 'TuningParams',
86| self.sesMgmt,
87| [[ 'invalidationTimeout', value ]]
88| )
89| AdminConfig.save()
90| text = 'The TuningParams object has ' + \ 91| 'been created successfully' 92|
except :
93| text = 'A problem was encountered while ' + \ 94| 'creating the TuningParams
object.' 95| else :
96| try :
97| AdminConfig.modify(
98| self.tuningParms,
99| [[ 'invalidationTimeout', value ]] 100| )
101| AdminConfig.save()
102| text = 'Update successful.'
103| except :
104| text = 'A problem was encountered while ' + \ 105| 'updating the
TuningParams object.' 106| self.message.setText( text )
```

It’s important to realize that this version of the script does a very bad thing. What is that, you ask? Well, take a look at the update(...) method again. It is the event handler code that verifies the user-specified input and attempts to modify the

invalidation timeout attribute for the administration console application.

The big mistake is the fact that the event handler routine contains calls to wsadmin scripting object (AdminConfig) methods. These kinds of calls should always be done on a separate (SwingWorker) thread. Why? It's because they are likely to require a non-trivial amount of time (the first call took about five seconds) to complete the requested action and return. In the meantime, the application can't respond to any user events. That's why it is not a good way to implement GUI scripts.

What does this look like when you run the script? Figure 19-8 shows a sample image created by this script. Go ahead and test it out yourself. See what happens when you enter a value and press Enter. Is there any kind of indication on the GUI that something is happening? Nope, at least not until the calls to the AdminConfig objects are complete and control returns to the application. Is this how you want your GUI applications to work and interact with your users? Absolutely not; this kind of behavior is unacceptable.

Figure 19-8. *Output from consoleTimeout_02.py*

Adding SwingWorker Instance to the Mix

Listing 19-19 shows how easy it is to take the statements from the update(...) method shown in consoleTimeout_02.py (that's Listing 19-18) and put them into an separate SwingWorker class. It also shows how simple the event handler update(...) method then becomes (see lines 122 and 122).

Listing 19-19. The WSAStask Class from consoleTimeout_03.py

```
10|class WSAStask( SwingWorker ) :
11| def __init__( self, app ) :
12| self.app = app # application reference
13| self.messageText = ''
14| SwingWorker.__init__( self )
15| def doInBackground( self ) :
16| problem = 'A problem was encountered while %s ' + \
```

```
17| 'the TuningParams object.'
18| messageText = self.messageText
19| self.app.textField.setEnabled( 0 )
20| self.app.message.setText(
21| '<html>working...' + ( ' ' * 20 )
22| )
23| value = self.app.textField.getText() # JTextField value
24| if not re.search( re.compile( '^\d+$' ), value ) :
25| messageText = 'Invalid numeric value: "%s"' % value
26| else :
27| if not self.app.tuningParms :
28| try :
29| AdminConfig.create(
30| 'TuningParams',
31| self.app.sesMgmt,
32| [[ 'invalidationTimeout', value ]]
33| )
34| AdminConfig.save()
35| messageText = 'The TuningParams object' + \
36| 'has been created successfully'
37| except :
38| messageText = problem % 'creating'
39| else :
40| try :
41| AdminConfig.modify(
42| self.app.tuningParms,
43| [[ 'invalidationTimeout', value ]]
44| )
45| AdminConfig.save()
46| messageText = 'Update complete.'
47| except :
48| messageText = problem % 'updating'
49| def done( self ) :
50| self.app.textField.setEnabled( 1 )
51| self.app.message.setText( self.messageText )
52|class consoleTimeout_03( java.lang.Runnable ) :
| ...
121| def update( self, event ) :
122| WSAStask( self ).execute()
```

One thing that you should notice when this SwingWorker thread is performing an update is the fact that application text input field is disabled. This kind of attention to detail adds significantly to the user experience and should be considered essential when you are developing graphical applications. Additionally, a status message is displayed to show that the thread is executing. It is only when this separate processing is complete that the text input field is enabled and an updated status message is displayed.

Adding Menu Items

As you've previously seen, you can add significant usability to your applications with simple menu items. Figure 19-9 shows the impact of making some small changes to this script.

Figure 19-9. *Images from consoleTimeout_04.py*

Listing 19-20 shows the new and modified code from the application class in this updated script. The menuBar() method creates and populates the application menu bar. Three new methods were added to respond to each of the new menu actions. And that is pretty much it. All in all, this is a good return on investment for about 50 lines of code. I think that you'll agree that the additional user value is significant.

In case you're wondering, the about(...)and notice(...) method adds some nice functionality. The former serves the same kind of role normally performed by a usage message that a non-graphical application uses to tell users how the program should be used, and the latter is useful as a valuable disclaimer message.

Listing 19-20. New and Modified Code from consoleTimeout_04.py

```
87|class consoleTimeout_04( java.lang.Runnable ) :
| ...

110| def run( self ) :
```

```
| ...
119| frame.setJMenuBar( self.menuBar() )
| ...
178| frame.setVisible( 1 )
179| def Exit( self, event ) :
180| sys.exit( 0 )
181| def about( self, event ) :
182| text = __doc__.replace( '<', '&lt;'
183| ).replace( '>', '&gt;'
184| ).replace( ' ', ' '
185| ).replace( '\n', '<br>' )
186| JOptionPane.showMessageDialog( 187| self.frame,
188| JLabel(
189| '<html>' + text, 190| font = monoFont 191| ),
192| 'About',
193| JOptionPane.PLAIN_MESSAGE 194| )
195| def notice( self, event ) : 196| JOptionPane.showMessageDialog( 197|
self.frame,
198| disclaimer,
199| 'Notice',
200| JOptionPane.WARNING_MESSAGE 201| )
202| def update( self, event ) : 203| WSAStask( self ).execute()
```

Changing from JFrame to JInternalFrame

For the next iteration, you'll see how to convert the script to use internal frames. Begin by taking a look at the revised consoleTimeout class from the revised consoleTimeout_05.py script. Listing 19-21 shows the important methods from this class. As you saw earlier, internal frames need to be added to an instance of the JDesktopPane class. Overall, it shouldn't be terribly new to you. You may, however, wonder about the call to the setSelected(...) method on line 186.

This isn't an error. Table 19-3 shows that the JDesktopPane class has a similar method called setSelectedFrame(...). This is a method from the JInternalFrame class, and the parameter is a boolean value indicating that the specified object instance is being selected (in which case, a value that is recognized as true should be specified). What happens if you comment out this line in the script and execute it?

The internal frame doesn’t obtain the focus, which means that the current timeout value isn’t obtained and the input field remains empty. At least until you put focus on the internal frame, such as by clicking somewhere on it. Then you see that the input field is populated with the current timeout value.

Listing 19-21. Revised consoleTimeout Class from consoleTimeout_05.py, Part 1

```
167|class consoleTimeout_05( java.lang.Runnable ) :
168| def run( self ) :
169| self.frame = frame = JFrame(
170| 'consoleTimeout_05',
171| size = ( 428, 474 ),
172| locationRelativeTo = None,
173| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
174| )
175| frame.setJMenuBar( self.MenuBar() )
176| desktop = JDesktopPane()
177| if globals().has_key( 'AdminConfig' ) :
178| self.timeout = self.initialTimeout()
179| self.inner = TextField( self ) # JTextField only
180| desktop.add( self.inner )
181| else :
182| self.inner = self.noWSAS() # WebSphere not found 183| desktop.add(
self.inner )
184| frame.add( desktop )
185| frame.setVisible( 1 )
186| self.inner.setSelected( 1 )
```

The initialTimeout() method (see line 178) isn’t shown because it contains the same steps that you’ve seen before to obtain the initial value, if one exists. The most important change in this script is shown in Listing 19-22. It contains the class used by all of the subsequent internal frame instances for this application. The first time you look at it, however, you may wonder why it is descended from InternalFrameListener and not from InternalFrameAdapter. That way, you wouldn’t need to have empty methods (ones like internalFrameClosed(...) that consist of a single pass statement).

Listing 19-22. InternalFrame Class from consoleTimeout_05.py, Part 1

```
90|class InternalFrame( JInternalFrame, InternalFrameListener ) :
91| def __init__( self,
92| title,
93| outer,
94| size,
95| location = None,
96| layout = None
97| ) :
98| if location == None :
99| location = Point( 0, 0 )
100| if layout == None :
101| layout = FlowLayout()
102| JInternalFrame.__init__(
103| self,
104| title,
105| 0, # resizeable = false
106| 0, # closable = false
107| size = size,
108| internalFrameListener = self,
109| layout = layout
110| )
111| self.setLocation( location ) # keyword parm doesn't exist 112| self.outer =
outer # application object 113| def internalFrameActivated( self, e ) :
114| self.outer.inner = e.getInternalFrame()
115| def internalFrameClosed( self, e ) :
116| pass
117| def internalFrameClosing( self, e ) :
118| pass
119| def internalFrameDeactivated( self, e ) :
120| self.outer.inner.message.setText( " )
121| def internalFrameDeiconified( self, e ) :
122| pass
123| def internalFrameIconified( self, e ) :
124| pass
125| def internalFrameOpened( self, e ) :
126| pass
127| def getValue( self ) :
128| print 'InternalFrame.getValue() - not yet implemented' 129| return None
130| def setValue( self, value ) :
```

131| print 'InternalFrame.setValue() - not yet implemented' 132| def working(self) :
133| print 'InternalFrame.working() - not yet implemented' 134| def finished(self) :
135| print 'InternalFrame.finished() - not yet implemented'

Listing 19-23 shows what happens when you try to base an InternalFrame class on the JInternalFrame and the InternalFrameAdapter classes. Jython complains about this issue. Why doesn't it complain about basing a class on JInternalFrame and the InternalFrameListener classes? This is allowed because the InternalFrameListener is actually an interface, not a class that can be instantiated.

Listing 19-23. Multiple Inheritance Issue

```
wsadmin>from javax.swing import JInternalFrame
wsadmin>from javax.swing.event import InternalFrameAdapter wsadmin>
wsadmin>class InternalFrame( JInternalFrame, InternalFrameAdapter ) :
wsadmin> def __init__( self, title ) :
wsadmin> JInternalFrame.__init__( self, title )
wsadmin>
WASX7015E: Exception running command: ""; exception information:

com.ibm.bsf.BSFException: exception from Jython:
Traceback (innermost last):
File "<input>", line 1, in ?
TypeError: no multiple inheritance for Java classes:
javax.swing.event.InternalFrameAdapter and javax.swing.JInternalFrame
wsadmin>
```

Listing 19-24 shows the TextField class from the consoleTimeout_05.py script. In it, you can see that the constructor (in lines 137-154) is responsible for populating the internal frame and providing the input components. Additionally, it includes instruction for the methods that are needed by this object. It is important to note that the internalFrameActivated(...) method, which is invoked when the object instance is activated, is responsible for updating the application attribute value that identifies the active internal frame, obtains the current timeout value from the application, and calls the object setValue(...) method to initialize the timeout value for this object instance.

Listing 19-24. TextField Class from consoleTimeout_05.py

```
136|class TextField( InternalFrame ) :
137| def __init__( self, outer ) :
138| InternalFrame.__init__(
139| self,
140| 'TextField',
141| outer,
142| size = ( 180, 85 ),
143| location = Point( 5, 5 )
144| )
145| self.add( JLabel( 'Timeout (minutes):' ) ) 146| self.text = self.add(
147| JTextField(
148| 3,
149| actionPerformed = outer.update 150| )
151| )
152| self.message = self.add( JLabel() ) 153| self.setVisible( 1 )
154| self.text.requestFocusInWindow() 155| def internalFrameActivated( self, e )
: 156| self.outer.inner = e.getInternalFrame() 157| self.setValue(
self.outer.timeout ) 158| def getValue( self ) :
159| return self.text.getText()
160| def setValue( self, value ) :
161| self.value = value
162| self.text.setText( value )
163| def working( self ) :
164| self.text.setEnabled( 0 )
165| def finished( self ) :
166| self.text.setEnabled( 1 )
```

All this is possible because of the framework provided by the InternalFrame class on which it is based. There are some important things to note about this class and how it fits into the application. One of these is the way in which this constructor begins by calling its base class constructor to initialize itself. One of the important steps performed by the base class constructor (InternalFrame.__init__) call, shown on lines 138-144, is the fact that it saves a reference to the application frame in an instance attribute called self.outer (see lines 90-110 in Listing 19-22).

What purpose does this application instance variable serve? It allows each inner

frame to easily obtain access to this reference so that the event handlers can manipulate the appropriate fields. How does it do this? Take a look at the internalFrameActivated(...) method in lines 155-157. When an internal frame is activated, the application variable called self.outer is used to update the self.inner application variable, which identifies the active inner frame. By defining and updating this variable, the utility routines can easily access the "current" or "active" internal frame values with simple statements. All you have to do is ensure that every internal frame class implements these values the same way.

For example, this reference (self.inner) allows the object instance to specify the application update(...) method as the ActionListener event handler for the text field on this inner frame. This application update(...) method, as shown in Listing 19-25, is extremely simple. As an event handler routine, it should be simple.

Listing 19-25. The update(...) Method from consoleTimeout_05.py

```
272| def update( self, event ) :
273| self.timeout = self.inner.getValue()
274| WSAStask( self ).execute()
```

There are, however, some subtle changes needed in the WSAStask class in this script that is instantiated to create a separate thread to modify the timeout value. Listing 19-26 shows this modified WSAStask class from the consoleTimeout_05.py script. The important changes are all related to the fact that this class needs to work with the active internal frame that initiated the creation of the task.

Take a look at line 51 where a local variable, frame, is used to hold a reference to the active inner frame. This reference is then used to invoke the appropriate working() method (line 52), the appropriate getValue() method (line 53), the appropriate message.setText(...) method (lines 54-56 and 89), and the appropriate finished() method (line 88).

The rest of this chapter shows how easily you can create other classes to take advantage of this design simplification.

Listing 19-26. WSAStask Class from consoleTimeout_05.py

```
46|class WSAStask( SwingWorker ) :
```

```
47| def __init__( self, app ) :
48| self.app = app # application reference
49| self.messageText = ''
50| SwingWorker.__init__( self )
51| def doInBackground( self ) :
52| messageText = self.messageText
53| frame = self.app.inner # Active Inner Frame
54| frame.working()
55| value = frame.getValue()
56| frame.message.setText(
57| '<html>working...' + ( ' ' * 20 )
58| )
59| if not re.search( re.compile( '^\d+$' ), value ) :
60| messageText = 'Invalid numeric value: "%s"' % value
61| else :
62| TPobj = 'TuningParams'
63| success = 'The %s object was created successfully.'
64| problem = 'A problem was encountered %s the %s object.'
65| if not self.app.tuningParms :
66| try :
67| self.tuningParms = AdminConfig.create(
68| TPobj,
69| self.app.sesMgmt,
70| [[ 'invalidationTimeout', value ]]
71| )
72| AdminConfig.save()
73| messageText = success % TPobj
74| except :
75| messageText = problem % ( 'creating', TPobj )
76| else :
77| try :
78| AdminConfig.modify(
79| self.app.tuningParms,
80| [[ 'invalidationTimeout', value ]]
81| )
82| AdminConfig.save()
83| messageText = 'Update complete.'
84| except :
85| messageText = problem % ( 'updating', TPobj )
```

```
86| def done( self ) :
87| frame = self.app.inner
88| frame.finished()
89| frame.message.setText( self.messageText )
```

What does this mean as far as the application's appearance is concerned? Figure 19-10 shows part of the application output created by this script. The remainder of this chapter deals with iterations of this script that will fill in the blanks as far as this application output is concerned.

Figure 19-10. *Output from consoleTimeout_05.py*

Adding a Second Internal Frame Class

Until now, it really looks like a lot of code is required to create a fairly simple application with one internal frame. Was it worth it? In order to answer that question, take a look at what is required to add a second, different, internal frame. Listing 19-27 shows is the code required to define another class—the TextandButton class, which is based on the TextField class shown in Listing 19-24. It is almost trivial due to the fact that all of the class methods, other than the constructor, are in fact identical to those defined in the base TextField class. Isn't inheritance wonderful? You bet it is!

Listing 19-27. The TextandButton Class from consoleTimeout_06.py

```
168|class TextandButton( TextField ) :
169| def __init__( self, outer ) :
170| InternalFrame.__init__(
171| self,
172| 'TextField and Button',
173| outer,
174| size = ( 180, 125 ),
175| location = Point( 5, 95 )
176| )
```

```
177| self.add( JLabel( 'Timeout: (minutes)' ) )
178| self.text = self.add(
179| JTextField(
180| 3,
181| actionPerformed = outer.update
182| )
183| )
184| self.button = self.add(
185| JButton(
186| 'Update',
187| actionPerformed = outer.update
188| )
191| self.setVisible( 1 )
192| self.text.requestFocusInWindow() 193|class consoleTimeout_06(
java.lang.Runnable ) : 194| def run( self ) :
|...
207| desktop.add( TextandButton( self ) )
```

This listing also shows, in line 207, the code needed to instantiate an object using this class and add it to the application desktop. Figure 19-11 shows the initial output of this iteration of the script. It's important to notice that you now have two internal frames, only one of which is active.

Figure 19-11. *Output from consoleTimeout_06.py*

Adding a Third Internal Frame Class

Let’s see what it takes to add a very different internal frame classes to this application. Listings 19-28, 19-29, and 19-30 show another class based on the TextField class that adds a group of RadioButtons for commonly used timeout values. In addition, an “other”RadioButton is defined that allows the users to specify an uncommon value using an associated text field.

Note, however, that this class demonstrates a solution to an issue that may not be obvious without explanation. In Listing 19-28, line 228, you can see that the self.setting variable is defined and initialized in the class constructor. As the comment indicates, this value is used in the stateChange(...) method (found in Listing 19-30) and the setValue(...) method (shown in Listing 19-29). The purpose of this variable is to deal with a possible race condition that might occur. It is possible that the stateChange(...) method will be invoked before the setValue(...) method has completed. If this occurs, the state of the text field may be indeterminate, so the stateChange(...) method exits without making any changes if the value of self.setting is non-zero.

This is the kind of thing that can occur in event-driven applications. So you should be aware of this possibility and be on the lookout for it. You may want to see what happens to the script without this and other issues related to interrupt-driven applications.

This is a good time for me to bring up an “oops” moment. At this point, while testing the current script, I found a significant flaw. You can see what happens with the previous version of the script—consoleTimeout_06.py. I haven’t gone back to correct either of the previous iterations, just to give you a chance to see the flaw and compare the current version of the script to see how it handles things better.

What’s the error? It’s most obvious in the previous script because of the presence of multiple inner frames. Enter an invalid numeric value in the text input field and press Enter. A message is displayed indicating that the value is bad. So far, so good. Now put focus on the other inner frame and see what happens. The bad value is displayed as the current value in the selected inner frame. Is this that you want to happen? I don’t think so. This is exactly the kind of thing that you need to consider when using inner frames that can gain and lose focus at unexpected times.

So, how do you fix it? Well, you have to make some changes to a few of the

existing methods. The biggest change needs to be made in the setValue(...) methods so that only valid values are saved. Note that the setValue(...) method only exists in the TextField class. The TextAndButton class reuses the base method, thereby limiting and simplifying the necessary changes.

It is important to note, however, that the new RadioButtons class has to implement this method because of the different data components in this new inner frame. The presence of the “other” radio button associated with a text input field means that you also have to change the update(...) method to properly deal with invalid input values.

Take a look at the resulting changes, focusing first on the new RadioButtons class. Listing 19-28 shows the constructors (__init__()) and getValue(...) methods. Notice how these are more complicated because they use RadioButtons and a text input field to display the current value.

One thing that is all too easy to overlook with Jython Swing code is how easily event handler methods can be assigned. Take another look at the class constructor, specifically line 210. This line uses a keyword argument to identify the ChangeListener event handler to each RadioButton object as the stateChange(...) method. The code for this method is shown in Listing 19-30.

Listing 19-28. RadioButtons Class from consoleTimeout_07.py, Part 1

```
205|class RadioButtons( TextField ) :
206| def __init__( self, outer ) :
207| InternalFrame.__init__(
| ...
213| )
214| self.add( JLabel( 'Timeout (minutes):' ) )
215| buttons = {}
216| self.bg = ButtonGroup()
217| for name in '0,15,30,60,Other'.split( ',' ) :
218| button = JRadioButton(
219| name,
220| itemStateChanged = self.stateChange
221| )
222| self.bg.add( button )
223| self.add( button )
```

```
224| buttons[ name ] = button
225| self.r00 = buttons[ '0' ]
226| self.r15 = buttons[ '15' ]
227| self.r30 = buttons[ '30' ]
228| self.r60 = buttons[ '60' ]
229| self.rot = buttons[ 'Other' ]
230| self.text = self.add(
231| JTextField(
232| '',
233| 3,
234| actionPerformed = outer.update
235| )
238| self.setting = 0 # see stateChange() and setValue() 239| self.setVisible( 1 )
240| def getValue( self ) :
241| if self.r00.isSelected() :
242| result = '0'
243| elif self.r15.isSelected() :
244| result = '15'
245| elif self.r30.isSelected() :
246| result = '30'
247| elif self.r60.isSelected() :
248| result = '60'
249| elif self.rot.isSelected() :
250| result = self.text.getText()
251| try :
252| int( result )
253| except :
254| messageText = badNumber % result
255| self.message.setText( messageText )
256| else :
257| result = None
258| return result
```

Listing 19-29 shows the setValue(...) method, which is also a bit more complex because it deals with multiple components to set the console timeout value.

Listing 19-29. setValue(...) Method in the RadioButtons Class from consoleTimeout_07.py

```
259| def setValue( self, value ) :
```

```
260| self.setting = 1
261| if value == '0' :
262| self.r00.setSelected( 1 )
263| self.r00.requestFocusInWindow()
264| self.text.setText( " )
265| self.text.setEnabled( 0 )
266| elif value == '15' :
267| self.r15.setSelected( 1 )
268| self.r15.requestFocusInWindow()
269| self.text.setText( " )
270| self.text.setEnabled( 0 )
271| elif value == '30' :
272| self.r30.setSelected( 1 )
273| self.r30.requestFocusInWindow()
274| self.text.setText( " )
275| self.text.setEnabled( 0 )
276| elif value == '60' :
277| self.r60.setSelected( 1 )
278| self.r60.requestFocusInWindow()
279| self.text.setText( " )
280| self.text.setEnabled( 0 ) 281| else :
282| self.rot.setSelected( 1 ) 283| self.text.setText( value ) 284|
self.text.setEnabled( 1 ) 285| self.text.requestFocusInWindow() 286| self.value =
value
287| self.setting = 0
```

You may note that this setValue(...) method doesn’t bother to include code to check for an invalid value in the text field. How can you get away with this? Isn’t this an oversight? Not really, because the update(...) method already includes that kind of code, so you’re covered.

Listing 19-30 shows the rest of the methods from this class. Note how easily you can enable and disable all of the RadioButton components using a simple loop that iterates over the list of RadioButton components.
Note also line 312, where the RadioButtons inner frame is instantiated and added to the desktop frame.

Listing 19-30. Utility Methods for the RadioButtons Class from consoleTimeout_07.py

```
288| def working( self ) :
289| for obj in [
290| self.r00, self.r15, self.r30,
291| self.r60, self.rot, self.text
292| ] :
293| obj.setEnabled( 0 )
294| def finished( self ) :
295| for obj in [
296| self.r00, self.r15, self.r30,
297| self.r60, self.rot
298| ] :
299| obj.setEnabled( 1 )
300| self.text.setEnabled( self.rot.isSelected() )
301| def stateChange( self, event ) :
302| item = event.getItem()
303| if not self.setting :
304| if item.getText() == 'Other' :
305| self.text.setEnabled( item.isSelected() )
306| else :
307| self.text.setEnabled( 0 )
308| self.text.setText( '' )
309| value = self.getValue()
310| if value :
311| self.outer.update( event )
312|class consoleTimeout_07( java.lang.Runnable ) :
313| def run( self ) :
| ...
327| desktop.add( RadioButtons( self ) )
```

I think that the functionality added by just over 100 lines of code in this class is pretty impressive. I hope that you agree. The most important point about this is that defining the way in which the classes interact and use the framework makes each additional class significantly easier to implement.

What does this look like as far as the application output is concerned? Well, take a look at Figure 19-12, which should help you understand the component configuration.

Figure 19-12. *Output from consoleTimeout_07.py*

Filling in the Blanks

Rather than continue showing every simple InternalFrame class definition, I'll cut to the chase[12] and simply show the end result of adding a few more internal frame descendant classes. The complete source is found in the consoleTimeout_08.py script file. Figure 19-13 shows the initial appearance of the final result of the application.

[12]See http://en.wikipedia.org/wiki/Cut_to_the_chase.

Figure 19-13. *Output from consoleTimeout_08.py*

Remember that this is merely a sample use of internal frames. Your applications are much more likely to have a variety of internal frame types, depending on your requirement. This sample is simply provided to demonstrate some of the ways in which an application that uses internal frames can deal with issues.

Summary

Internal frames can make your applications dynamic and interesting. There is a tradeoff in that events may contribute to increased complexity, especially when the frames need to share data in some way. This should have been quite obvious in the chapter example, even though only one data value needed to be shared between the inner frames. You get to decide if the increased complexity is worth

the investment of time and effort. In the next chapter, you'll use the same iterative approach to build an application that interacts with the WebSphere scripting object to create an application that can be used to more easily display the help text for the objects and their methods.

Chapter 20

Building a Graphical Help Application

In Chapter 15, you learned how to retrieve information from a web page and use Swing objects to find and display only the details in which you were interested. Now you're going to use the same kind of techniques to build an application that allows you to more easily find and manipulate the Help text that is available for the wsadmin scripting objects.

In Chapter 7 of the book titled *WebSphere Application Server Administration Using Jython*,[1,2] you learned about the wsadmin scripting objects. It describes how to use these objects and explained how to use their help(...) methods to obtain information about the methods that each object provides. Unfortunately, the interactive sessions and sample scripts provided in that book aren't as useful as they could be if an interactive graphical application were available. In this chapter, you'll remedy that situation.

Showing the Help Text

To begin, consider what you need to display the output returned by a call to the Help.help() method. You'll need to keep these issues in mind:

• You should display the text in a scrollable pane
• If you want the text displayed correctly, don't use a proportional font

Only a tiny amount of effort is required to produce the first iteration, the main class of which is shown in Listing 20-1. One of the things that I hadn't remembered was that the Help text contains a number of tab characters (see '\t') to align the text. To simplify things, the text of the call to the help() method is immediately passed to the string's expandtabs() method, which is provided by Jython (see line 19).

Listing 20-1. WSAShelp_01.py Help.help() in a Scrollable Pane

```
9|class WSAShelp( java.lang.Runnable ) :
10| def run( self ) :
11| frame = JFrame(
12| 'WSAShelp',
13| locationRelativeTo = None,
14| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 15| )
16| frame.add(
17| JScrollPane(
18| JTextArea(
19| Help.help().expandtabs(),
20| 20,
21| 80,
22| font = Font( 'Courier' , Font.PLAIN, 12 ) 23| )
24| )
25| )
26| frame.pack()
27| size = frame.getSize()
28| loc = frame.getLocation()
29| loc.x -= ( size.width >> 1 )
30| loc.y -= ( size.height >> 1 )
31| frame.setLocation( loc )
32| frame.setVisible( 1 )
```

[1]See http://www.ibmpressbooks.com/store/websphere-application-server-administration-using-jython-9780137009527. [2]Hereafter referred to as “the WAuJ book” for simplicity’s sake.

The other statements that may cause you some pause are found in lines 27-31. When the frame in constructed (lines 11-15), the locationRelativeTo keyword argument places the upper-left corner of the frame in the center of the screen. Lines 27-31 adjust the frame to the left by half the frame width (line 29) and up by half the frame height (line 30) to center the frame in the screen. Figure 20-1 shows the simple result of the WSAShelp_01.py script.

Figure 20-1. *WSAShelp_01.py sample output*

Using a Tabbed Pane

This next iteration uses tabbed panes to display the help text of the five wsadmin scripting objects (Help, AdminApp, AdminConfig, AdminControl, and AdminTask). Listing 20-2 demonstrates how a simple list (lines 28-34) can be used to easily identify the name of the tab and the associated scripting object about which help should be displayed.

Listing 20-2. Modifications to the WSAShelp Class in WSAShelp_02.py

```
11|class WSAShelp( java.lang.Runnable ) :
| ...
28| objs = [
29| ( 'Help' , Help ),
30| ( 'AdminApp' , AdminApp ),
31| ( 'AdminConfig' , AdminConfig ),
32| ( 'AdminControl', AdminControl ),
33| ( 'AdminTask' , AdminTask )
```

```
34| ]
35| tabs = JTabbedPane()
36| for name, obj in objs :
37| tabs.addTab(
38| name,
39| JScrollPane(
40| JTextArea(
41| obj.help().expandtabs(),
42| 20,
43| 90,
44| font = Font( 'Courier' , Font.PLAIN, 12 )
45| )
46| )
47| )
48| frame.add( tabs )
| ...
```

Figure 20-2 shows sample output of the tabs produced by this script. Using this script helps you not only verify the way that it looks and responds, but it's also a good place to stop and think about what you can and should do next.

Figure 20-2. *WSAShelp_02.py sample output*

What can you learn from this script? The first thing is that the text area should be read-only, which is a trivial change. The next is that I wondered how difficult it would be to replace the textArea with a split pane. The top portion could display the general help for the scripting object and the bottom portion could scroll the method help. You'll see what that looks like before you have to decide if you want to continue down that path or use a different approach.

Adding Split Panes

In order to split the scripting object help text, you need to determine where the general description ends and the list of methods begins. To do that, you will use a regular expression (RegExp). This topic is discussed in the WAuJ book in Chapter 7. For all of the scripting objects (except the AdminTask object), you can use a relatively simple RegExp to determine where the method names start in the help text. The AdminTask help is a different beast entirely, so you have to deal with that in a different way. Let's start by dealing with the four scripting objects for which a relatively simple regular expression pattern works.

Listing 20-3 shows a simple script that demonstrates using a RegExp to determine the length of the description section for the majority of scripting objects.

Listing 20-3. The desc.py Script

```
1|import re
2|pat = re.compile( r'^(\w+)(?:\s+.*)$', re.MULTILINE )
3|objs = [
4| ( 'Help' , Help ),
5| ( 'AdminApp' , AdminApp ),
6| ( 'AdminConfig' , AdminConfig ),
7| ( 'AdminControl', AdminControl )
8|]
9|print ' Object | #Lines | 1st method'
10|print '-------------+--------+--------------------'
11|for name, obj in objs :
12| text = obj.help()
13| mo = re.search( pat, text )
14| desc = text[ :mo.start( 1 ) ].strip().splitlines() 15| method = text[ mo.start( 1 )
: mo.end( 1 ) ] 16| print '%-12s | %6d | %s' % ( name, len( desc ), method )
```

Figure 20-3 shows the output of the desc.py script in Listing 20-3. The WSAShelp_03.py script incorporates this technique to process these scripting objects so that their tabs will contain a split pane.

```
    Object    | #Lines | 1st method
-------------+--------+--------------------
Help          |     16 | attributes
AdminApp      |     15 | deleteUserAndGroupEntries
AdminConfig   |     19 | attributes
AdminControl  |     21 | completeObjectName
```

Figure 20-3. *Output generated by desc.py*

Figure 20-4 shows the next version of this WSAShelp script, which uses this technique. In this version of the script, you can see that all of the tabs, except for the AdminTask tab, show a vertical split pane. The scripting object description is above the divider, and the information about the scripting object methods is below. The divider even has one touch expand icons (the little triangles) that can be used to expand or collapse the corresponding top or bottom pane, all with a single click. This is starting to look pretty neat. The complete WSAShelp_03.py script can be found in the code\Chap_20 directory.

Figure 20-4. *Image from WSAShelp_03.py*

Text Highlighting

Looking at the output of WSAShelp_03.py made me wonder how difficult it would be to add text highlighting to the application. You could use this to locate specific text on a page. Before you change the WSAShelp script, let's see what is required to do this.

First, you need to realize that the JTextArea class doesn't provide a way to do this kind of thing. It, along with JTextField and JPasswordField, are subclasses of the abstract JTextComponent class,[3] which allows only a single kind of attribute for the entire component. This means that all of the data in the component must have the same font and color.

What you want to do is use a JTextComponent that allows multiple attributes for the contents. This means that you need to use JEditorPane or JTextPane[4] to hold and display the application's text.
To decide if you want to add this capability to your WSAShelp script, start simply by creating a simple script that allows you to improve your understanding of what might be required to use this capability well.
Figure 20-5 shows the output of Highlight.py; its complete source is found in the code\Chap_20 directory. The image shows the text of the Help.wsadmin() method, displayed in a read-only JTextPane that is within a ScrollPane. Beneath the text pane is a label and an input field that can be used to enter the highlighted text. The scroll pane above the input field shows an image after the word "the" has been entered,[5] and the scroll pane was scrolled down to display a number of highlighted occurrences of the word.

Figure 20-5. *Output of a simple script showing text highlighting*

Listing 20-4 shows part of the Highlight class from the Highlight.py script used to create the images shown in Figure 20-5. The first part of the Highlight class (lines 16-26) contains the center(...) method, which is called in line 51, to position the frame in the center of the screen.

[3] See http://docs.oracle.com/javase/8/docs/api/javax/swing/text/JTextComponent.html.
[4]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JTextPane.html.
[5]Remember that the JTextField ActionListener is called when the user presses Enter.

Lines 32-34 instantiate an instance of the static DefaultHighlightPainter class.[6] This allows the application to specify the color to be used to highlight portions of the JTextPane.
Lines 35-40 instantiate the JTextPane, make it read-only, define the preferred size, initialize the text, set the font, and provides the default highlighter to be used.[7]
Line 41 is needed so that when the application becomes visible, the start of the text pane area is seen. If this statement is commented out or removed, the end of

the text will be shown instead. Line 42 is where the text pane is added to the application frame within a scroll pane.
Lines 43-49 should look familiar to you. This is where the label and text input field are defined and added to a panel, which is then added to the bottom of the application frame. Of particular note is the keyword assignment on line 46, which identifies the ActionListener event handler that is invoked when the user presses Enter while the keyboard focus is on this component.

Listing 20-4. Portions of the Highlight.py Script

```
15|class Highlight( java.lang.Runnable ) :
| ...
27| def run( self ) :
28| frame = JFrame(
29| 'Highlight',
30| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
31| )
32| self.painter = DefaultHighlighter.DefaultHighlightPainter(
33| Color.YELLOW
34| )
35| pane = self.tPane = JTextPane(
| ...
40| )
41| pane.moveCaretPosition( 0 )
42| frame.add( JScrollPane( pane ), 'Center' )
43| info = JPanel( BorderLayout() )
44| info.add( JLabel( 'Find text:' ), 'West' )
45| tf = JTextField(
46| actionPerformed = self.search
47| )
48| info.add( tf, 'Center' )
49| frame.add( info, 'South' )
50| frame.pack()
51| self.center( frame )
52| frame.setVisible( 1 )
53| tf.requestFocusInWindow()
```

Listing 20-5 shows the search(...) method from the Highlight.py script. This is where the actual highlighting works occurs. It begins by retrieving the contents

of the input (JTextField) and removing any existing highlights. The advantage of doing it this way is that if the user enters an empty string, pressing the Enter key will remove any existing highlights.

[6]See http://docs.oracle.com/javase/8/docs/api/javax/swing/text/DefaultHighlighter.Def
[7]See http://docs.oracle.com/javase/8/docs/api/javax/swing/text/DefaultHighlighter.htm

Lines 54-70 are where the searching and adding of highlights to the text pane occurs. The string find(...) method, as shown in lines 62 and 70, return the offset or position in the text string of the data for being located. If no match is found, an offset of -1 is returned. So, if a non-negative offset is returned, it identifies the location of the first character of the data string in the text string being searched. The actual work of creating the highlighter occurs in lines 64-68 and identifies the painter to be used to highlight the text in which the user is interested.

Listing 20-5. The search(...) Method from the Highlight.py Script

```
54| def search( self, event ) :
55| data = event.getSource().getText()
56| hiliter = self.tPane.getHighlighter()
57| hiliter.removeAllHighlights()
58| if data :
59| doc = self.tPane.getDocument()
60| text = doc.getText( 0, doc.getLength() )
61| start = 0
62| here = text.find( data, start )
63| while here > -1 :
64| hiliter.addHighlight(
65| here,
66| here + len( data ),
67| self.painter
68| )
69| start = here + len( data )
70| here = text.find( data, start )
```

Adding Text Highlighting to WSAShelp Application

Overall, that appears to be pretty simple and straightforward, right? Is there

anything that you need to consider or be concerned about in order to add this capability to the existing WSAShelp application?

One issue that comes to mind is how the user specifies the text to be highlighted. Do you want to add an input field below the tabbed pane or consider something else? Might you want to add some menu options that allow you to highlight the text?

How should the highlighting work? As you can see in the Highlight.py script, the DefaultHighlighter class instance is associated with the abstract JTextComponent class using the setHighlighter(...) method or using the Jython keyword assignment syntax, as shown in line 45 of Listing 20-4. It isn't clear whether one of these class instances can, or should be, shared among JTextPane instances. This isn't good practice, so I, for one, will avoid doing this. What does this mean for your application? Well, for each of the text pane instances that you want to highlight, you need a highlighter instance.

How should the application deal with highlighted text when the user changes the selected tab? That's going to make things interesting, isn't it?
You can start by adding a ChangeListener to the TabbedPane. What does that take? As is frequently the case, it's much easier than I originally thought it would be. Listing 20-6 shows the trivial changes that you need to make to the WSAShelp_03.py script in order to determine and display the name of the selected tab. The WSAShelp_04.py script includes these minor modifications.
Listing 20-6. Changes Need to Identify Tab Selection from WSAShelp_04.py

```
| ...
39| tabs = JTabbedPane( stateChanged = self.tabPicked )
| ...
89| def tabPicked( self, event ) :
90| pane = event.getSource()
91| print pane.getTitleAt( pane.getSelectedIndex() )
```

With this ChangeListener method, you can easily access the name of the selected tab. If you add a dictionary to the application class that is indexed by tab name, with each entry being a tab-specific highlighter instance, the change listener method can use this information to highlight the tab-specific text. Let's hold off on this decision for a moment while you learn about some complications.

Tabbed Highlighting Complications

What is the application supposed to do when the user selects a tab that contains a split pane? If you want the text in each pane to be highlighted, that is going to require a separate highlighter for each panel, isn't it? Well, as you saw with the Highlight.py application, you can use the DefaultHighlighter for the text pane. Can't you use this same technique for each of the panes on each of the tabs in the application? If you do, the change listener event handler can process each of the panes on the tab to reset and populate the highlights to be displayed when the tab is selected. This would probably work best if you had a single, shared "find" input field.

During the development and testing of this iteration, it became apparent how quickly things can get complicated. What do I mean? Well, consider this for a few moments. In the change listener, you can easily determine which tab was selected. You can also tell what the tab contains. Unfortunately, it will either be a JScrollPane or a JSplitPane. The JSplitPane will have two parts, each of which will contain a JScrollPane. So far, so good. Now comes some of the "fun" (challenging) part. Given a JScrollPane, you can access the actual JTextPane component by calling the getViewport().getView() methods from the JScrollPane component. I'm sure you can see the circuitous kind of path that needs to be followed to access the JTextPane in which you're interested.

For a change listener method called when the tab contains a JScrollPane and not a JSplitPane, the methods shown in Figure 20-6 need to be called just to get access to the JTextPane contained within the JScrollPane, on the specified tab. To simplify the code as you'll as improve its performance, you'll create a dictionary indexed by the tab name, with references to each of the JTextPanes contained on the specified tab.

```
pane  = event.getSource()
index = pane.getSelectedIndex()
comp  = pane.getComponentAt( index )
tPane = comp.getViewport().getView()
```

Figure 20-6. *Calls needed to find a JTextPane in a JScrollPane*

Based on this observation, changes you're made to produce the WSAShelp_05.py script. Figure 20-7 shows an image from this application after "default" is highlighted on the AdminConfig tabs.

Figure 20-7. *WSAShelp_05.py sample output*

To create this iteration of the application required about 50 more lines of code, most of which can be seen in Listing 20-7. The highlight(...) method is called by the change listener event handler (the tabPicked(...) method), which is invoked when the user selects a new tab. Additionally, it is called by the action listener event handler (the lookFor(...) method) when the user presses Enter while the keyboard focus is on the JTextField at the bottom of the window.

Listing 20-7. New and Changed Methods in WSAShelp_05.py

```
35| def hilight( self, tPane, text ) :
36| hiliter = tPane.getHighlighter()
37| hiliter.removeAllHighlights()
38| if text :
```

```
39| doc = tPane.getDocument()
40| info = doc.getText( 0, doc.getLength() )
41| start = 0
42| here = info.find( text, start )
43| while here > -1 :
44| hiliter.addHighlight(
45| here,
46| here + len( text ),
47| self.painter
48| )
49| start = here + len( text )
50| here = info.find( text, start )
51| def lookFor( self, event ) :
52| text = event.getSource().getText()
53| index = self.tabs.getSelectedIndex()
54| name = self.tabs.getTitleAt( index )
55| for tPane in self.tPanes[ name ] :
56| self.hilight( tPane, text )
| ...
134| def tabPicked( self, event ) :
135| pane = event.getSource()
136| index = pane.getSelectedIndex()
137| name = pane.getTitleAt( index )
138| try :
139| for tPane in self.tPanes[ name ] :
140| self.hilight( tPane, self.textField.getText() ) 141| except :
142| pass
```

■ **Note** You may wonder why you need a try/except clause in the change listener method. it is there because this method is invoked when the first tab is added to the JTabbedPane container. When this occurs, the data structure (the self.tPanes dictionary) hasn't been initialized, so the specified reference (self.tPanes[name]) is invalid. other than that, the changes are pretty straightforward and make the application more useful, at least in my opinion.

Displaying Methods in a Table

I don't know about you, but every time that I look at the method (and scripting objects) sections (the bottom split pane sections), I wonder how difficult it would

be to display this part in a table within a scroll pane. The big question related to this change is how it affects the highlighting.

Let's approach these challenges one at a time. You can start by creating a table for the bottom part of the split panes. Then, you can take a look at the cell rendering portion to see if you can easily resolve the highlighting issue.
So how should you go about building a table containing the method name in one column and the method abstract/description in another? Let's start by using a simple application that does this and only this. Then, you can determine what changes, if any, need to be made before incorporating this into your application. Figure 20-8 shows the method names and descriptions that you're extracted from the output of calling the Help.help() method.

Method	Description / Abstract
attributes	given an MBean, returns help for attributes
operations	given an MBean, returns help for operations
constructors	given an MBean, returns help for constructors
description	given an MBean, returns help for description
notifications	given an MBean, returns help for notifications
classname	given an MBean, returns help for classname
all	given an MBean, returns help for all the above
help	returns this help text
AdminControl	returns general help text for the AdminControl object
AdminConfig	returns general help text for the AdminConfig object
AdminApp	returns general help text for the AdminApp object
AdminTask	returns general help text for the AdminTask object
wsadmin	returns general help text for the wsadmin script launcher
message	given a message id, returns explanation and user action message

Figure 20-8. *Proof of concept for displaying method info in a table*

Listing 20-8 shows the majority of the statements from the MethodTable1.py proof of concept (PoC) script used to produce the output shown in Figure 20-8. Since it is a PoC script, it's simple. As you can see, the run(...) method is very simple. It uses the parseMethodHelp(...) method to extract the method names and descriptions from the specified help text in order to populate the table.

Listing 20-8. run(...) Method of MethodTable Class from MethodTable1.py Script

```
10|class MethodTable1( java.lang.Runnable ) :
```

```
| ...
22| def run( self ) :
23| frame = JFrame(
24| 'MethodTable1',
25| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
26| )
27| helpText = Help.help().expandtabs()
28| headings = [ 'Method', 'Description / Abstract' ]
29| data = self.parseMethodHelp( helpText )
30| table = JTable(
31| data,
32| headings,
33| font = Font( 'Courier' , Font.PLAIN, 12 )
34| )
35| frame.add( JScrollPane( table ), 'Center' )
36| frame.pack()
37| self.center( frame )
38| frame.setVisible( 1 )
```

The parseMethodHelp(...) method, shown in Listing 20-9, is where all of the real work takes place. It uses a regular expression (RegExp) to locate each method name in the specified input string.

Listing 20-9. parseMethodHelp(...) Method of the MethodTable Class from the MethodTable1.py Script

```
39| def parseMethodHelp( self, helpText ) :
40| def fix( text ) :
41| text = text.replace( '\n', ' ' ).strip()
42| return re.sub( ' +', ' ', text )
43| methRE = re.compile( r'^(\w+)(?:\s+.*)$', re.MULTILINE )
44| result = []
45| mo = methRE.search( helpText )
46| name = None
47| while mo :
48| start, finish = mo.span( 1 )
49| if name :
50| result.append(
51| [
52| name,
```

```
53| fix( helpText[ prev : start ] )
54| ]
55| )
56| name = helpText[ start : finish ]
57| prev = finish + 1
58| mo = methRE.search( helpText, finish )
59| if name :
60| result.append( [ name, fix( helpText[ prev: ] ) ] )
61| return result
```

The remainder of the text up to the next method name is considered the associated description. The result of this method is an array, with one row for each method, and two columns. The first column of the table contains the method name and the second column the entire description string. In order to do this, the fix routine, shown in lines 40-42 of Listing 20-9, replaces the newline characters ('\n') with a space. Then, all of the leading, trailing, and multiple adjacent spaces are removed.

Highlighting Text Within the Table

One of the nice things about using a JTextPane component to hold the text to be displayed is that you can use the DefaultHighlighter as you'll as the DefaultHighlightPainter class instances to locate and highlight text of interest to the user. Unfortunately, these classes aren't available on the contents of the JTable instance. So, what can you do? The default renderer for the table cells uses a JLabel instance to format the cell contents.

The neat thing about this is that a JLabel can use HTML tags to format the cell contents. This means that all you have to do to highlight a portion of a cell is surround it with the appropriate HTML tags. To do that, make the changes shown in Listing 20-10.

Listing 20-10. Changes to the Highlight Table Text from MethodTable2.py

```
10|class MethodTable2( java.lang.Runnable ) : | ...
22| def run( self ) :
| ...
29| data = self.parseMethodHelp( helpText )
30| for r in range( len( data ) ) :
```

```
31| for c in range( len( data[ r ] ) ) : 32| data[ r ][ c ] = self.hiliteText( 33| data[ r ]
[ c ],
34| 'help'
35| )
36| table = JTable(
37| data,
38| headings,
39| font = Font( 'Courier' , Font.PLAIN, 12 ) 40| )
| ...
45| def hiliteText( self, text, findWord ) : 46| return '<html>' + text.replace(
47| findWord,
48| '<font bgcolor=yellow>%s</font>' % findWord 49| )
```

Figure 20-9 shows the sample output generated by this script. By default, the word “help” is highlighted.

Figure 20-9. *PoC for displaying highlighted table text*

Using Tables in the Help Application

Are there any problems with adding this to the help application? If you take a few moments to test the MethodTable scripts, you are likely to see a noticeable delay during the application startup. Unfortunately, this is the kind of thing that users find particularly annoying. As discussed earlier, whenever you encounter an operation that could take a long time, it should execute on a separate thread. Since the operation in question involves the creation of the attribute and

description table, it should be executed on a separate thread.

These choices might cause you to ask yourself some questions. For example, what should you display on the bottom part of the split pane until the table thread completes? An easy and reasonable approach is to display a simple message indicating that results will be available shortly. So, you'll just use a JLabel instance with a message.

You might also wonder which attributes the tables should use. You don't want the user to attempt to make any changes to the table data, so the table cells should be read-only. Additionally, you know that the data type for each cell should be a java.lang.String. Add a table model class (methodTableModel) that descends from the DefaultTableModel class.

Listing 20-11. The methodTableModel Class from WSAShelp_06.py

```
26|class methodTableModel( DefaultTableModel ) :
27| def __init__( self, data, headings ) :
28| DefaultTableModel.__init__( self, data, headings )
29| def isCellEditable( self, row, col ) :
30| return 0
31| def getColumnClass( self, col ) :
32| return String
```

Listing 20-11 shows just how simple this kind of descendent class can be. Unfortunately, you also need to add code to adjust the table column widths. Without it, the table will give each column as close to 50% of the table width as it can. With the information that you want to display, this isn't a great distribution of the available space. So, WSAShelp_06.py also includes the setColumnWidths(...) method shown in Listing 20-12 to try to adjust the column widths.

Listing 20-12. The setColumnWidths(...) Method from WSAShelp_06.py

```
76| def setColumnWidths( self, table ) :
77| header = table.getTableHeader()
78| tcm = table.getColumnModel() # Table Column Model
79| data = table.getModel() # To access table data
80| margin = tcm.getColumnMargin() # gap betyouen columns
81| rows = data.getRowCount() # Number of rows
```

```
82| cols = tcm.getColumnCount() # Number of cols
83| for i in range( cols ) : # For col 0..N
84| col = tcm.getColumn( i ) # TableColumn: col i
85| idx = col.getModelIndex() # model index: col i
86| render = col.getHeaderRenderer()# header renderer
87| if render :
88| comp = render.getTableCellRendererComponent(
89| table,
90| col.getHeaderValue(),
91| 0,
92| 0,
93| -1,
94| i
95| )
96| cWidth = comp.getPreferredSize().width
97| else :
98| cWidth = -1
99| for row in range( rows ) :
100| val = str( data.getValueAt( row, idx ) ) 101| r = table.getCellRenderer( row, i
)
102| comp = r.getTableCellRendererComponent( 103| table,
104| val, # formatted value 105| 0, # not selected 106| 0, # not in focus 107| row,
# row num 108| i # col num 109| )
110| cWidth = max(
111| cWidth,
112| comp.getPreferredSize().width 113| )
114| if cWidth > 0 :
115| col.setPreferredWidth( cWidth + margin )
```

Figure 20-10 shows a sample image from the WSAShelp_06.py script. As you can see, some of the tabs can’t display complete information that is available to both columns. When this occurs, the contents have ellipses (. . .) appended to the truncated values.

Figure 20-10. *Sample output from WSAShelp_06.py*

Grab and drag the separator bar between the column headings and see what happens when you change the column widths. Is that responsive enough for you? I don't notice any delay at all and am quite pleased.

Fixing the Table Appearance

Many people would find the ellipses a distraction and an annoyance. How do you fix this? The first thing to do is to remember how the cell data is represented. In Chapter 12, you learned about cell renderers. Fortunately, all of the table cells contain simple strings. So you only need to provide a renderer for this data type.

The challenge, though, is how you deal with the cell data that doesn't fit in the available space. First, you need to be able to figure out:

• How much space is available
• How you want to display the data
• How you split the data to fit in the available space

The program uses a monospace font, which makes sense when you're displaying the help text that is normally seen in an interactive wsadmin session.
Another choice that makes life a little easier is to use a JTable to display the method information. How does it make things easier? If you take another look back at Chapter 12, you'll see that the default cell renderer for a JTable will use a JLabel to display a cell. You can then use HTML to format the cell contents. This allows you to specify
 (the HTML line break tag) where you want the break to occur. Unfortunately, if you are going to use multiple lines of text to display the cell text, you also have top deal with the row height to allow multiple lines.
An additional challenge involves how you intend to split any long method names. Blanks aren't allowed as part of a method name, so you have to decide if you want to split the method name between arbitrary letters or use a little additional effort and recognize that method names use a camelcase[8] naming convention. Personally, I prefer to split the method names between the camelcase words. You need to use a JLabel method, specifically the getFontMetrics(...) method, to determine how much of the data will fit in the available width.
Listings 20-13 and 20-14 show the table cell renderer class from WSAShelp_07.py. Unfortunately, it doesn't fit on one page so has to be shown in two pieces.

Listing 20-13. The methRenderer Class from WSAShelp_07.py Part 1

```
class methRenderer( DefaultTableCellRenderer ) :
 def __init__( self ) :
 self.fm = JLabel().getFontMetrics( monoFont )
 self.widths = [ 0, 0 ]
 self.hiText = ''
 def getTableCellRendererComponent(
 self,
 table, # JTable - table containing value
 value, # Object - value being rendered
 isSelected, # boolean - Is value selected?
 hasFocus, # boolean - Does this cell have focus?
 row, # int - Row # ( 0 .. N-1 )
```

39| col # int - Col # (0 .. N-1)
40|) :
41| def camelWords(name) :
42| prev, result = 0, []
43| for i in range(len(name)) :
44| ch = name[i]
45| if ch == ch.upper() or ch == '_' : 46| result.append(name[prev:i]) 47| prev = i
48| result.append(name[prev:])
49| return result
50| DTCR = DefaultTableCellRenderer
51| comp = DTCR.getTableCellRendererComponent(52| self, table, value, isSelected, hasFocus, row, col 53|)
54| pWidth = self.widths[col] # Preferred column width

[8]See http://en.wikipedia.org/wiki/CamelCase.

The first portion of the class, shown in Listing 20-13, shows the class constructor in lines 28-31 as well as the initial portion of the getTableCellRendererComponent(...) method. Note that this method includes the local camelWords(...) method on lines 41-49. It is only used in the next segment of code (on line 62) where the contents of column 0 are being processed.

It takes advantage of the similarity of splitting method names and the method abstract text. If the text doesn't fit into the available width, the data is reformatted using HTML to use multiple lines. Since the renderer is the class that uses HTML to format the data, it is also the right place to add HTML to highlight any text that the user requests.

If you look closely at the renderer class in Listing 20-14, you might notice a couple of methods that are unique to this renderer (they don't exist in the base class). These are the setHiText(...), and setWidths(...) methods on lines 92-95. These were added to provide the class instance with the information that it needs to format the text to fit in the available width, as well as add the appropriate highlighting, as needed. Is this a perfect answer? Probably not, but it is certainly good enough, especially for this application.

Listing 20-14. The methRenderer Class from WSAShelp_07.py Part 2

```
55| if pWidth : # Has it been set?
56| hiHTML = '<font bgcolor=yellow>%s</font>'
57| pWidth -= 3
58| if self.fm.stringWidth( value ) > pWidth :
59| if col :
60| pad, words = ' ', value.split( ' ' )
61| else :
62| pad, words = '', camelWords( value )
63| result, curr = '<html>', ''
64| for word in words :
65| width = self.fm.stringWidth(
66| curr + pad + word
67| )
68| if width > pWidth :
69| result += curr + '<br>'
70| curr = ''
71| if curr :
72| curr += pad + word
73| else :
74| curr = word
75| result += curr
76| if self.hiText :
77| if result.count( self.hiText ) > 0 : 78| result = result.replace( 79| self.hiText,
80| hiHTML % self.hiText 81| )
82| comp.setText( result )
83| else :
84| if self.hiText :
85| if value.count( self.hiText ) > 0 : 86| value = value.replace( 87| self.hiText,
88| hiHTML % self.hiText 89| )
90| comp.setText( '<html>' + value ) 91| return comp
92| def setHiText( self, text ) :
93| self.hiText = text
94| def setWidths( self, width0, width1 ) :
95| self.widths = [ width0, width1 ]
```

How and when would the setWidths(...) method need to be called? Think about it for a few moments. When does the table need to be displayed? The first time would be when the tab on which a table exists is selected. Another time would be when the application frame was resized (maximized). If you take a look at the

WSAShelp_07.py script, and search for calls to this routine, you will find calls to this routine in these two event handlers.

Take a few moments to test this script and see what happens when you resize the frame. Personally, I like the way it responds to the resize requests; I hope that you agree.

Selecting Table Cells

Now that you have the scripting object methods in a nice little table at the bottom of each split pane, what's next? You can now progress to one of the goals that I had for this application. I wanted to be able to select the tab for a specific scripting object, scroll down to show the name of the method in which I was interested, and select the method. Wouldn't it be neat if selecting the method in the table showed the help for the selected method? What is needed to do this? Listing 20-15 shows the class that was added to provide this capability to the WSAShelp_08.py script.

Listing 20-15. ListSelectionListener from WSAShelp_08.py

```
30|class cellSelector( ListSelectionListener ) :
31| def __init__( self, table, WASobj ) :
32| self.table = table
33| self.WASobj = WASobj
34| self.objName = WASobj.help()[ :40 ].split( ' ' )[ 2 ]
35| def valueChanged( self, event ) :
36| if not event.getValueIsAdjusting() :
37| table = self.table
38| row = table.getSelectedRow()
39| if row > -1 :
40| method = table.getModel().getValueAt( row, 0 ) 41| title = '%s.help( "%s" )'
% ( 42| self.objName,
43| method
44| )
45| text = self.WASobj.help( method ) 46| text = JTextArea(
47| text,
48| 20,
49| 80,
50| font = monoFont
```

```
51| )
52| dialog = JDialog(
53| None, # owner 54| title, # title 55| 1, # modal = true 56| layout =
BorderLayout(),
57| locationRelativeTo = None 58| )
59| dialog.add(
60| JScrollPane(
61| text
62| ),
63| BorderLayout.CENTER
64| )
65| dialog.pack()
66| dialog.setVisible( 1 )
```

The class constructor may require a bit of explanation though. The reason for providing the table as an argument to the constructor will be explained shortly. The reason for having the WASobj (AdminApp, AdminConfig, and so on), specified makes things for the valueChanged(...) method much simpler, because given the scripting object, and the name of the method in question, all the routine needs to do is invoke the help(...) method, as see on line 45. This leaves you with the expression on line 34. What is it doing?

Listing 20-16 shows how the constructor expression works. Basically, it depends on the fact that the first line of the help text for each of these scripting objects uses the same format. The message number is followed by a sentence that starts with “The”, followed by the name of the scripting object. The expression on line 34 of Listing 20-15 simply uses this fact to save the name of the specified scripting object.

Listing 20-16. Extracting an Object Name from its Help Text

```
wsadmin>for obj in [ AdminApp, AdminConfig, AdminControl, Help ] :
wsadmin> print obj.help()[ :40 ]
wsadmin>
WASX7095I: The AdminApp object allows ap
WASX7053I: The AdminConfig object commun
WASX7027I: The AdminControl object enabl
WASX7028I: The Help object has two purpo
wsadmin>
```

Instances of the cellSelector listener class shown in Listing 20-15 are added when the table is created in the tableTask instance, specifically in the done(...) method. Listing 20-17 shows how the selection listener is added to table using its selection model. The fact that the selection listener instance is added to the selection model, and not the table itself, is the main reason that the table instance is passed on the cellSelector constructor. The code in the valueChanged(...) event handler method, as shown in Listing 20-15, uses the table to identify the user-selected method, so that the appropriate help text can be displayed.

Listing 20-17. Adding Selection Listener Instance from WSAShelp_08.py

```
178| table.getSelectionModel().addListSelectionListener(
179| cellSelector(
180| table,
181| self.WASobj
182| )
183| )
```

What do these changes allow you to do? Figure 20-11 shows a sample dialog box that is displayed when some of the scripting object table rows are selected. Note how the title of the dialog box identifies the scripting object and method for which help is being displayed.

Figure 20-11. *Sample image from WSAShelp_08.py script*

Note to make things simpler for the application, modal dialog boxes are used. additionally, the text is displayed in a scroll pane, just in case it's larger than the available space.

Adding a Menu

One of the things that I wasn't too happy about with the previous scripts was how the "Highlight text" label and input field were always showing on the bottom of the frame. Another alternative is to use menu items for this and other kinds of actions. Let's start simple and replace the label and input field with a "Show" menu that allows the user to specify text to be highlighted. You can also provide some information about the script with some "Help" menu items. What does this do to the application? Figure 20-12 has some images from the WSAShelp_09.py script, showing the initial look of the menu and the associated drop-down menus. Additionally, you see an image showing the input dialog box that is displayed when the user selects the Show ➤ Highlight Text menu item.

Figure 20-12. *Sample images from WSAShelp_09.py script*

The complete application can be found in the code\Chap_20\WSAShelp_09.py file and requires only about 80 additional lines of code to add this non-trivial functionality.

AdminTask.help('-commands')

In the WAuJ book two different scripts were used to format all of the help text for the wsadmin scripting objects. That's because the AdminTask scripting object is significantly different from the other scripting objects. In fact it provides a framework for WebSphere developers to add scripting capabilities by adding methods to the AdminTask framework, instead of forcing changes to be made to the existing objects.

When I began to consider adding the ability of displaying AdminTask help information to the WSAShelp scripts, I first considered adding some kind of listener to the AdminTask pane that would allow the user to select either the "help -commands", or "help -commandGroups" portions of that pane. Unfortunately, this would not be an obvious solution to the problem. How would

you convey this information to the user? Maybe by using some kind of special highlighting, possibly even using something like the convention used by browsers to indicate that text is, in fact, a link to additional information. I quickly discarded this approach after taking another look at the menu items that had just been added to the WSAShelp_09.py script. This appeared to be a much more obvious solution to the problem.

Unfortunately, trying to display information about the available AdminTask commands brings challenges of its own. For example, depending on the version of the WebSphere Application Server product being used, there can be a huge list of available AdminTask commands from which to choose. The following sections address some of the questions that come to mind.

How Do We Find and Identify Existing Commands?

The obvious answer to this question is to take a look at the output produced by the AdminTask.help('-commands') method call. How much text is generated by this call? As mentioned earlier, it depends. To give you an idea, you can run a quick test. Listing 20-18 shows that for a WebSphere version 7.0, network deployment installation, almost 1,200 lines of text are generated.

Listing 20-18. Lines of Text from the AdminTask.help('-commands') Output

```
wsadmin>print len( AdminTask.help( '-commands' ).splitlines() ) 1190
wsadmin>
```

WebSphere version 8.0 has more than 1,250 lines and WebSphere version 8.5 has more than 1,400. The time to process this output is not something that you would want to do in a simple event handler. That would be likely to cause a noticeable and unacceptable delay in the application. As you’ve read, you should consider doing this processing on a separate (SwingWorker) thread.

Showing the User that Something Is Happening

Since the processing of the AdminTask.help('-commands') command output may cause a non-trivial delay, you need to consider how you communicate this fact to the user. One thing that came to mind was that you can initialize the Show ➤ AdminTask ➤ -commands menu item as disabled, at least until the processing is complete and the data exists in a usable form.

If you proceed with this decision, the SwingWorker class that you're going to use has to know the menu item that needs to be enabled once processing completes. Additionally, it needs to know how to make the results of its processing available to the application.

It seems pretty obvious that the result of the processing of this text should be an array or list of AdminTask command names. Don't you think?
Alright, what's it going to take to have a separate SwingWorker thread to process the output of the AdminTask.help('-commands') call? Listing 20-19 shows that it doesn't take much effort. The doInBackground(...) method makes the call and processes each line of the resulting text. It can use a simple regular expression to identify the method names and each one that is found is added to the list. Then, all the done(...) method needs to do is enable the specified menu item.

Listing 20-19. ATcommandTask Class from WSAShelp_10.py

```
176|class ATcommandTask( SwingWorker ) :
177| def __init__( self, List, menuItem ) :
178| SwingWorker.__init__( self )
179| self.commands = List
180| self.menuItem = menuItem
181| def doInBackground( self ) :
182| data = AdminTask.help( '-commands' ).splitlines()
183| for line in data[ 1: ] :
184| mo = re.match( '([a-zA-Z_2]+) -', line ) 185| if mo :
186| self.commands.append( mo.group( 1 ) ) 187| def done( self ) :
188| self.menuItem.setEnabled( 1 )
```

Now that you have this long list of commands or methods, how do you make things easy for your users? Scrolling through a list of 1,000 plus names is not exactly user friendly.
Do you remember what you did back in Chapter 15, with the javadocInfo script? You added an input field that allowed the user to filter the list of items to be displayed.[9] You can do the same thing here. If I'm looking for the AdminTask methods that have something to do with a particular topic, it can be somewhat daunting to look through 1,000 plus command names. By adding a filtering mechanism, you can greatly diminish the number of commands through which the user has to look.
Figure 20-13 shows some sample images from the WSAShelp_10.py script. The

first shows what happens to the application when the “Show” menu item has been selected. The second shows what happens when the AdminTask menu item has been selected. The third shows the initial view of the dialog box. One thing that you should notice is the size of the scrollbar thumb, which gives you an indication of the size of the list. Finally, you see what happens to the number of commands that contain “Lis”. I think that you’ll agree that this technique vastly improves the user experience when dealing with this huge list of AdminTask commands.

Figure 20-13. *Sample images from WSAShelp_10.py script*

[9]The filtering was first added to javadocInfo_06.py, which is discussed in the section entitled “Filtering the List.”

One thing that you may notice about the WSAShelp_10.py script output is what happens when you select Show ➤ AdminTask ➤ -commandGroups menu item. A placeholder dialog box is displayed, indicating that work remains to be done.

AdminTask.help(‘-commandGroups’)

Now you get to figure out how to deal with the “-commandGroups” menu item. When you were working with the “-commands” menu item, it made sense to generate a list of the available AdminTask commands. With this selection, that makes less sense. Since each commandGroup is supposed to represent a group of similar commands, doesn’t it make more sense to display this as a tree, with each node of the tree representing one of the command groups? That way, to see the commands within a group, you would only have to expand the group by selecting (double-clicking) on a group name.

Like the ATcommandTask class seen in Listing 20-19 it seems reasonable to expect that you’re going to need another SwingWorker descendent class to process the AdminTask.help('-commandGroups') output to build the application dialog box.

The first thing that this class will need to do is to extract the commandGroup names from the AdminTask.help('-commandGroups') output. To figure out how to do this, first take a look at what this output looks like. Listing 20-20 shows the first few lines of output produced by this command.

Listing 20-20. The AdminTask.help('-commandGroups') Output wsadmin>print AdminTask.help('-commandGroups') WASX8005I: Available admin command groups:

AdminAgentNode - Admin Agent Managed Node related tasks
AdminAgentSecurityCommands - Commands used to configure security ...
AdminReports - Admin configuration reports
AdministrativeJobs - This command group contains all the job mana...
AppManagementCommands - Application management commands.
AuditAuthorizationCommands - Audit Authorization Table Commands ...

This format allows for easy identification and processing of the command group names. When a group name is identified (AdminAgentNode), then a subsequent AdminTask.help(...) method call can be used to identify the associated commands, if any, that exist within the group. For example, the output seen in Listing 20-21 shows the help text for two different command groups, only one of which (AdminReports) has associated commands. The format used by these command group help text messages make processing of them easy as well.

Listing 20-21. AdminTask.help(...) Using commandGroup Names

wsadmin>print AdminTask.help('AdminAgentNode') WASX8007I: Detailed help for command group: AdminAgentNode
Description: Admin Agent Managed Node related tasks
Commands:
wsadmin>print AdminTask.help('AdminReports') WASX8007I: Detailed help for command group: AdminReports

Description: Admin configuration reports Commands:
reportConfigInconsistencies - Checks the configuation repository ...
reportConfiguredPorts - Generates a report of the ports configure...

wsadmin>

Listing 20-22 shows the SwingWorker descendent class used to process the AdminTask.help('-commandGroups') text, to identify the names of the command groups and the associated AdminTask commands. This class is from the WSAShelp_11.py script. It may be interesting to note that the constructor requires only one parameter, that is, the variable used to identify the menu item to be enabled when processing is complete.

Listing 20-22. ATgroupsTask Class from WSAShelp_11.py

```
196|class ATgroupsTask( SwingWorker ) :
197| def __init__( self, menuItem ) :
198| SwingWorker.__init__( self )
199| self.menuItem = menuItem
200| def doInBackground( self ) :
201| try :
202| data = AdminTask.help(
203| '-commandGroups'
204| ).expandtabs().splitlines()
205| self.root = DefaultMutableTreeNode(
206| 'command groups'
207| )
208| empty = []
209| for line in data[ 1: ] :
210| mo = re.match( '([a-zA-Z ]+) -', line )
211| if mo :
212| groupName = mo.group( 1 )
```

```
213| group = None
214| text = AdminTask.help( groupName )
215| cmds = text.find( 'Commands:' )
216| if cmds > 0 :
217| for line in text[cmds+9:].splitlines() :
218| mo = re.match('([a-zA-Z_2]+) -', line)
219| if mo :
220| if not group :
221| group = DefaultMutableTreeNode(
222| groupName
223| )
224| group.add(
225| DefaultMutableTreeNode(
226| mo.group( 1 )
227| )
228| )
229| if group :
230| self.root.add( group )
231| else :
232| empty.append( groupName ) 233| except :
234| print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ] 235| def done( self ) :
236| self.menuItem.setEnabled( 1 )
```

Notice that this class constructor didn't include a parameter for the result like you saw in the ATcommandTask class, in Listing 20-19. This was done simply to show another approach. Are there problems with this approach? To answer that question, let's take a look at the code used to instantiate this class and access the resulting data (a DefaultMutableTreeNode[10]) for the root node. Listing 20-23 shows the other places in the code that instantiate and execute the ATgroupsTask, as well as retrieve the result of its processing.

Listing 20-23. Use of ATgroupsTask class in WSAShelp_11.py

```
| ...
464| self.ATgroupsTree = None
465| self.groups = ATgroupsTask( groupMI )
466| self.groups.execute()
| ...
623| def showCmdGroups( self, event ) :
```

```
| ...
631| left = JPanel( layout = BorderLayout() )
632| left.add(
633| JScrollPane(
634| JTree(
635| self.groups.root,
636| rootVisible = 0,
637| valueChanged = self.pickATgroup
638| )
639| ),
640| BorderLayout.CENTER
641| )
| ...
```

Note it's important to note that in order to access the result of the ATgroupsTask processing, you need to have a variable for referencing the object. that is why WSAShelp_11.py includes the self.groups variable.

It might not be immediately obvious that the showCmdGroups(...) method uses the self.groups variable to reference the root node for the tree. The event listener routine is called when the Show ➤ AdminTask ➤ -commandGroups menu item is selected.

What does this mean as far as the JTree instance is concerned? To understand this, you should consider when and how often this routine is called. Each time the user invokes this routine, a new JTree instance is created. For a tiny application such as this, it might not make much of a difference, but it is better to understand the implications of your decisions as far as potential performance impact is concerned.

[10]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/DefaultMutableTreeNo

Another ATgroupsTask Implementation

If you want to have the JTree populated by the ATgroupsTask process, what do you need to do? First, you have to realize what is required in order for the ATgroupsTask instance to modify the data of a JTree parameter passed to it. The JTree documentation[11] shows that no method exists to replace the tree root (no setRoot(...) method exists). Why is this?

You have to remember that the JTree class, like many other Swing classes (the JTable), has the data managed by an associated model. In the case of the JTree class, the data exists in, and is maintained by, the associated TreeModel[12] instance, which in all likelihood would be a DefaultTreeModel[13] object instance. The DefaultTreeModel class is where the setRoot(...) method resides. So, what do you need to change in the ATgroupsTask? Not much. You need to add a parameter for the constructor to hold the JTree object to be updated. It then adds a statement in the done(...) method that gets the tree model instance and uses its setRoot(...) method to replace the tree data. Another encouraging thing about this approach is that it also simplifies the event handler, instead of creating a new JTree.

Figure 20-14 shows the output after selecting the commandGroups menu item and then double-clicking the AutoGen Commands entry. Note that if you only select a table entry, it isn't expanded to display the child nodes.

Figure 20-14. *Sample image from WSAShelp_12.py script*

[11] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JTree.html.
[12] See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/TreeModel.html.
[13] See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/DefaultTreeModel.htm

Step It Up: Displaying the "Steps" Help Text

I was going to finish this application, as well as the chapter, with the previous iteration of the script. But I demonstrated it to a friend who asked me some very good questions. What about the “steps” that some of the AdminTask commands have? Is there any way to view the help for these steps?

I told him that I would think about it and see what I could do. To begin, I had to see how many of the AdminTask commands had steps, as well as to see what it would take to display this information with a minimum impact to the existing script.

It didn’t take much effort to write a script to locate and display the names of the AdminTask commands that had any steps. It also displayed a little detail about the information that was processed. For example, when I ran this script (steps.py) in a WebSphere V 7 environment, it showed that 42 of the 1,162 commands have one or more steps (3.61%). That’s why I had forgotten all about this aspect of some AdminTask commands—only a small number have the “Steps” section populated.

How Should You Do It?

Realizing that steps exist doesn’t explain how to display the help text for a step. To understand that, you need only take a look at the AdminTask help text, as shown in Listing 20-24.

Listing 20-24. The Part of AdminTask.help() Text About Steps

help commandName stepName display detailed information for the specified step belonging to the specified command

This answers the question, “how do you get the help text for a command step?” But it doesn’t answer the more challenging question, “how do you modify the WSAShelp_12.py script to allow the user to see this help text?”

Should You Add a Menu?

One thought is to add a menu to the command window that would allow the users to get the help for a specific step. When the command help text is being processed, if no steps exist, then the menu could be disabled or you could decide not to add the menu. Each step could be listed on the menu, thereby allowing the users to select the step for which help could be displayed in a modal dialog box. The good news is that you know how to do these things and you’ve seen them done. However, it’s not clear whether this is the best approach available. Let’s

see what other options exist before you make a decision.

Should You Add Another Split Pane?

Another alternative that comes to mind to detect when steps exist and then use a split pane with the general command help description at the top, and a table with one row for each of the steps at the bottom. Again, this is something that you know how to do since this technique is shown on the middle tabs of the application. The bad news is that it would seem to be a significant amount of work and code for very infrequent use. Since less than 5% of the AdminTask commands have steps, it is not clear that this approach is worth the time and effort to implement.

Can You Make Parts of the Text Pane Selectable?

Another alternative is to allow the users to click on any of the visible step names. This is a little different and might be interesting to implement. Unfortunately, since you haven't done this yet, it's not clear how much time and effort, not to mention code, is required to do this.

And Now for Something Completely Different...

One of the things that got my attention while I was thinking about how best to approach this task was the fact that all of the previous options require extra effort on the part of the user to decide which of the steps they want to investigate. If they want to view multiple steps, multiple actions are required (select a menu item, look at the modal dialog box, close it, and repeat as necessary). Is there an easier way?

That's when I wondered about the possibility of displaying the help text for any and all steps in the same scrollable text pane containing the help text for the command. Another advantage to this approach is the fact that it requires about a dozen lines of code to be added in two places (the pickATcmd(...) and the pickATgroup(...) methods).

This iteration of the script is available in the code\Chap_20 directory, and is named WSAShelp_13.py. Figure 20-15 shows the "Steps" section of the help text for the createCluster command, and the beginning of the help text for the first of the associated steps.

Figure 20-15. *The help text for the clusterConfig step*

One of the important things about this iteration is that the first approach that came to mind wasn't the easiest or best approach for this particular situation. In fact, I considered multiple approaches and started to investigate what changes would be required for each approach before the "simple" answer came to me. Sometimes, it's not the first idea that comes to mind that ends up being the best approach.

Summary

All in all, this is now a pretty decent example of the iterative development of a non-trivial graphical application that uses a large variety of Swing capabilities. Even though it isn't perfect, it's a good tutorial on one way that Jython Swing applications can be developed. It is also a good example of how you might want to decide which approach is the most appropriate for you and your users' needs. Don't be afraid to stop and think about the options and which ones are going to make the application most useful for your users. It's also good to remember that you can do a quick proof of concept for potential enhancements for your application.

In the next chapter, you'll look at a different application and use an iterative improvement to add functionality as you improve the application. This application will be used to display details related to the security configuration report about a WebSphere environment.

Chapter 21

A Security Configuration Report Application

Not too long ago, someone asked me how difficult it would be to generate a security configuration report, similar to what is available from the WebSphere administration console. This chapter explores the creation of such an application to display the security report in a user-friendly way, all the while using the techniques that you've learned so far. This will allow the users to interact with the information in a way that provides a better understanding of the security-related configuration information for a WebSphere Application Server.

Generating the Administration Console Report

To get started, you are going to generate a static HTML security report for the environment. Then you will learn how, using iterative steps, to create an application that uses a GUI to display this security information in a user friendly way. To begin, an administrator must use the administration console to generate the report. This requires the administrator to be logged into the administration console using an appropriate security role. To generate the report, follow these steps (see Figure 21-1):

1. Select/expand the Security section on the left frame.
2. Select the Global Security link.
3. Click on the Security Configuration Report button.

Figure 21-1. *Generating a security configuration report*

This causes a browser page to be displayed containing the security report. Figure 21-2 shows the top of one such configuration report browser page.

Security Configuration Report - Mozilla Firefox

https://localhost:9055/ibm/console/com.ibm.ws.console.security.forwardCmd.do?forwardName=security.report.main&csrfid=-2026929473

Security Configuration Report

WebSphere Application Server Core Security settings for host name:ragweed . Report generated on:Jan 4, 2013, 09:47:22

Security Configuration Report

Console Name	Security Configuration Name	Value
Security Settings		
Active authentication mechanism	activeAuthMechanism	LTPA_1
User account repository	activeUserRegistry	WIMUser
Allow basic authentication	allowBasicAuth	true
Application security	appEnabled	false
Authentication cache timeout	cacheTimeout	600 seco
Default SSL settings	defaultSSLSettings	SSLConfi
Dynamically update run time when SSL configuration changes occur	dynamicallyUpdateSSLConfig	true
Administrative security	enabled	true
Restrict access to resource authentication data	enforceFineGrainedJCASecurity	false
Java 2 security	enforceJava2Security	false
Warn if applications are granted custom permissions	issuePermissionWarning	true
Use realm-qualified user names	useDomainQualifiedUserNames	false
Use the local security server	useLocalSecurityServer	true
Authentication mechanisms and expiration		

Figure 21-2. *Sample Security Configuration Report output*

Do you see any problem with this? My biggest complaint about this report is that it is very static. About the only thing that you can do with it is to use the text-searching capability of the browser to locate specific text. These steps were tested on three different versions of the WebSphere Application Server product and are the same for versions 7.0, 8.0, and 8.5, which is good. It makes it easier to describe it here. The biggest challenge that I encountered in verifying these steps in this version is the fact that I had to see which, if any, of the deployment managers were active, start those that weren't currently active, log into the administration console, and check the steps.

The next challenge was consuming the report data. The browser page produced when the report button is selected is, in a word, huge. The good news, however, is that it is in a table format, with each section having its own header row shaded in a different color. The report is so big because there are about two dozen

sections and some of the sections are quite large. In fact, some of the sections have more than 300 rows of information. In case you are interested, I've included the script that I used to determine this information—reportSectionSize.py—in the code\Chap_21 directory.

The Scripting Report Method

Since you're interested in accessing this information using an wsadmin script, you need to determine whether there is a command or scripting object method that allows you access to this information. You can even use the WSAShelp.py script from the previous chapter to locate it.

Choose the Show ➤ Highlight Text menu and move through the various tabs. You'll see that there isn't a method with a name or abstract that contains the word "report." Use the Show ➤ AdminTask ➤ -commands filtering mechanism to verify that the only commands that include "eport" (the "r" is omitted to avoid case-sensitive returns) are those shown in Figure 21-3.

Figure 21-3. *AdminTask commands containing "eport"*

Select the first option—generateSecConfigReport—and click on the top one-touch expansion icon on the divider. You'll see an image similar to Figure 21-4.

Figure 21-4. *AdminTask commands containing "eport"*

The unfortunate thing about this help text is that it doesn't tell you anything about the format of the report that is produced. To do that, you need to execute the command and take a look at the output. Fortunately, this is really easy to do with the wsadmin command. Because of the number of lines generated, I suggest that you use the wsadmin -c command-line option to execute a single command and have the output redirected to a file. Listing 21-1 shows an example of this wsadmin command.[1]

Listing 21-1. Sending the generateSecConfigReport to a File

```
wsadmin -conntype none -lang jython -c \
"print AdminTask.generateSecConfigReport()" >SecCfgRpt.txt
```

It doesn't take too much effort to understand the format of the data produced by this AdminTask command. The first few lines are shown in Figure 21-5. There are four columns of data on each line, and semicolons (;) delimit the column data. The second line of text corresponds to the column headings on the table (the rows containing Console Name and Security Configuration Name).

[1] The -lang jython command-line option is needed only if you haven't changed the default scripting language in the appropriat wsadmin.properties file. The -conntype none option tells wsadmin to execute in local mode, so no connection is attempted to a possibly inactive application server. It is important to remember that the command should appear on a single line. The backslash\isn't actually

part of the command and should not be entered.

```
DOMAIN ; Cell ; resourceName ; resourceType ;
Console Name ; Security Configuration Name ; Value ; Console Path Name ;
_Security Settings ;   ;   ;   ;
Active authentication mechanism ; activeAuthMechanism ; LTPA_1 ; …
```

Figure 21-5. *Example security report output*

Each section heading row has a leading underscore (_) in front of the section heading (such as _Security Settings). The rows that follow contain the values for each row in the table.[2]

First Attempt

Let's see what it takes to process this information using the data format that was just discussed, and see how it looks. Listing 21-2 shows the first (arguably quick and dirty) attempt at doing this.

Listing 21-2. SecConfigReport_01.py—Quick and Dirty Attempt

```
class SecConfigReport_01( java.lang.Runnable ) :
 def run( self ) :
 frame = JFrame(
 'SecConfigReport_01',
 size = ( 300, 300 ),
 locationRelativeTo = None,
 defaultCloseOperation = JFrame.EXIT_ON_CLOSE
 )
 data = []
 text = AdminTask.generateSecConfigReport()
 for line in text.splitlines()[ 2: ] :
 data.append(
 [ info.strip() for info in line.split( ';' ) ]
 )
 frame.add(
 JScrollPane(
 JTable( data, ';;;;'.split( ';' ) )
 )
 )
 frame.pack()
 frame.setVisible( 1 )
```

Figure 21-6 shows the output that is produced when this script is executed. The good news is that you don't have to invest too much time or effort into this application. The bad news is that this lack of effort shows. [2]In fact, this information is how the reportSectionSize.py script processes the data to determine the number and sizes of the report sections.

SecConfigReport

_Security Setti...				
Active authenti...	activeAuthMec...	LTPA_1	Security > Glo...	
User account ...	activeUserRe...	WIMUserRegi...	Security > Glo...	
Allow basic a...	allowBasicAuth	true	Security > Glo...	
Application se...	appEnabled	false	Security > Glo...	
Authentication...	cacheTimeout	600 seconds	Security > Glo...	
Default SSL s...	defaultSSLSet...	SSLConfig_1	Security > SS...	
Dynamically u...	dynamicallyU...	true	Security > SS...	
Administrative...	enabled	true	Security > Glo...	
Restrict acces...	enforceFineGr...	false	Security > Glo...	
Java 2 security	enforceJava2...	false	Security > Glo...	
Warn if applic...	issuePermiss...	true	Security > Glo...	
Use realm-qu...	useDomainQ...	false	Security > Glo...	
Use the local ...	useLocalSec...	true	Security > Glo...	
_Authenticatio...			Security > Glo...	
Authentication...	authConfig	system.KRB5	Security > Glo...	
Authentication...	authContextI...	com.ibm.ISec...	Security > Glo...	
Authentication...	authValidation...	system.KRB5	Security > Glo...	
Enable deleg...	enabledGssC...	true	Security > Glo...	
Kerberos conf...	krb5Config		Security > Glo...	
Kerberos keyt...	krb5Keytab		Security > Glo...	
Kerberos real...	krb5Realm		Security > Glo...	
Kerberos serv...	krb5Spn	WAS/${HOST}	Security > Glo...	
Simple authe...	simpleAuthCo...	system.KRB5	Security > Glo...	
Trim Kerbero...	trimUserName	true	Security > Glo...	

Figure 21-6. *Output of SecConfigReport_01.py*

From this output, you can see that each cell in the last column is empty. That's a little strange, isn't it? Based on this information, I took another, closer look at the text returned by the AdminTask command and found that every line ends with a semicolon, followed by a space (that's ";"). So, you can take this into account and remove this extraneous delimiting data in your subsequent attempts.

The other thing that this first attempt shows is that the time needed to process the text, although it wasn't insignificant, wasn't exactly terrible, at least on the machine that I was using. You will need to decide if the data processing should

be performed on a separate thread, and if so, what the application should display while this processing is occurring.

Second Attempt, Ignoring the Last Delimiter

This next attempt focuses on fixing those immediately obvious shortcomings. This requires two changes to the code. Listing 21-3 shows these modifications.
Listing 21-3. SecConfigReport_02.py—Modified Lines

```
| ...
17| for line in text.splitlines()[ 2: ] :
18| data.append(
19| [
20| info.strip() for info in
21| line[ :-2 ].split( ';' )
22| ]
23| )
24| frame.add(
25| JScrollPane(
26| JTable( data, ';;;'.split( ';' ) ) 27| )
28| )
| ...
```

1. Instead of splitting each line of text using the semicolon delimiter, you simply need to ignore the last two characters of each line. To understand this change, compare line 19 in Listing 21-2 with lines 19-22 in Listing 21-3. The only reason that this list is shown on multiple lines in Listing 21-3 is to fit better within the book's margins.

2. Since you are removing and ignoring the last delimiter from each line, you also need to remove one of the semicolons in the second parameter of the JTable instantiation. This is so the empty header array has the same number of columns as the data array parameter.

Figure 21-7 shows the resulting output of this application. Notice the empty last column that you saw in the first attempt is now gone.

SecConfigReport

_Security Settings			
Active authenticati...	activeAuthMechan...	LTPA_1	Security > Global ...
User account repo...	activeUserRegistry	WIMUserRegistry_1	Security > Global ...
Allow basic authe...	allowBasicAuth	true	Security > Global ...
Application security	appEnabled	false	Security > Global ...
Authentication cac...	cacheTimeout	600 seconds	Security > Global ...
Default SSL settin...	defaultSSLSettings	SSLConfig_1	Security > SSL ce...
Dynamically updat...	dynamicallyUpdat...	true	Security > SSL ce...
Administrative sec...	enabled	true	Security > Global ...
Restrict access to ...	enforceFineGrain...	false	Security > Global ...
Java 2 security	enforceJava2Sec...	false	Security > Global ...
Warn if application...	issuePermission...	true	Security > Global ...
Use realm-qualifi...	useDomainQualifi...	false	Security > Global ...
Use the local sec...	useLocalSecurity...	true	Security > Global ...
_Authentication m...			Security > Global ...
Authentication con...	authConfig	system.KRB5	Security > Global ...
Authentication con...	authContextImplC...	com.ibm.ISecurity...	Security > Global ...
Authentication vali...	authValidationCo...	system.KRB5	Security > Global ...
Enable delegation...	enabledGssCred...	true	Security > Global ...
Kerberos configur...	krb5Config		Security > Global ...
Kerberos keytab file	krb5Keytab		Security > Global ...
Kerberos realm n...	krb5Realm		Security > Global ...
Kerberos service ...	krb5Spn	WAS/${HOST}	Security > Global ...
Simple authentica...	simpleAuthConfig	system.KRB5	Security > Global ...
Trim Kerberos rea...	trimUserName	true	Security > Global ...

Figure 21-7. *Initial output of SecConfigReport_02.py*

Adding a Table Model and Cell Renderer

If you test the application, it shouldn't take long to figure out that, as discussed in Chapter 12, the default values used by the JTable class may not be best fit for this application. For example, is there any reason that you would want the user to be able to modify any of the table cells? Not if you want to create an application that can display security configuration settings.

Another default setting that doesn't work well here is the fact that the user can select multiple non-contiguous rows. Figure 21-8 shows this multiple selection issue at work.

SecConfigReport

_Security Settings			
Active authenticati...	activeAuthMechan...	LTPA_1	Security > Global ...
User account repo...	activeUserRegistry	WIMUserRegistry_1	Security > Global ...
Allow basic authe...	allowBasicAuth	true	Security > Global ...
Application security	appEnabled	false	Security > Global ...
Authentication cac...	cacheTimeout	600 seconds	Security > Global ...
Default SSL settin...	defaultSSLSettings	SSLConfig_1	Security > SSL ce...
Dynamically updat...	dynamicallyUpdat...	true	Security > SSL ce...
Administrative sec...	enabled	true	Security > Global ...
Restrict access to ...	enforceFineGrain...	false	Security > Global ...
Java 2 security	enforceJava2Sec...	false	Security > Global ...
Warn if application...	issuePermission...	true	Security > Global ...
Use realm-qualifi...	useDomainQualifi...	false	Security > Global ...
Use the local sec...	useLocalSecurity...	true	Security > Global ...
_Authentication m...			Security > Global ...
Authentication con...	authConfig	system.KRB5	Security > Global ...
Authentication con...	authContextImplC...	com.ibm.ISecurity...	Security > Global ...
Authentication vali...	authValidationCo...	system.KRB5	Security > Global ...
Enable delegation...	enabledGssCred...	true	Security > Global ...
Kerberos configur...	krb5Config		Security > Global ...
Kerberos keytab file	krb5Keytab		Security > Global ...
Kerberos realm n...	krb5Realm		Security > Global ...
Kerberos service ...	krb5Spn	WAS/${HOST}	Security > Global ...
Simple authentica...	simpleAuthConfig	system.KRB5	Security > Global ...
Trim Kerberos rea...	trimUserName	true	Security > Global ...

Figure 21-8. *Output of SecConfigReport_02.py showing selection issue*

It's also not great that the sections are hard to distinguish from the rest of the data. Maybe you could add a cell renderer to make these rows easier to find and see. Listing 21-4 shows the two classes that were added to this script for the table model and cell renderer instances.

Listing 21-4. SecConfigReport_03.py—TableModel and Cell Renderer

```
class reportTableModel( DefaultTableModel ) :
 def __init__( self, data, headings ) :
 DefaultTableModel.__init__( self, data, headings )
 def isCellEditable( self, row, col ) :
 return 0
 def getColumnClass( self, col ) :
 return String
```

```
19|class reportRenderer( DefaultTableCellRenderer ) :
20| def getTableCellRendererComponent(
21| self,
22| table, # JTable - table containing value 23| value, # Object - value being
rendered 24| isSelected, # boolean - Is value selected? 25| hasFocus, # boolean -
Does this cell have focus? 26| row, # int - Row # ( 0 .. N-1 ) 27| col # int - Col # (
0 .. N-1 ) 28| ) :
29| DTCR = DefaultTableCellRenderer
30| comp = DTCR.getTableCellRendererComponent(
31| self, table, value, isSelected, hasFocus, row, col 32| )
33| if value :
34| if table.getValueAt( row, 0 ).startswith( '_' ) : 35| if col == 0 :
36| value = value[ 1: ]
37| comp.setText( '<html><b>%s</b>' % value ) 38| return comp
```

Figure 21-9 shows the effects of these changes. This is a real improvement. I’m not sure if displaying the section row data in bold is good enough, but it does make it easier to locate these rows, doesn’t it?

SecConfigReport

Security Settings			
Active authenticati...	activeAuthMechan...	LTPA_1	Security > Global ...
User account repo...	activeUserRegistry	WIMUserRegistry_1	Security > Global ...
Allow basic authe...	allowBasicAuth	true	Security > Global ...
Application security	appEnabled	false	Security > Global ...
Authentication cac...	cacheTimeout	600 seconds	Security > Global ...
Default SSL settin...	defaultSSLSettings	SSLConfig_1	Security > SSL ce...
Dynamically updat...	dynamicallyUpdat...	true	Security > SSL ce...
Administrative sec...	enabled	true	Security > Global ...
Restrict access to ...	enforceFineGrain...	false	Security > Global ...
Java 2 security	enforceJava2Sec...	false	Security > Global ...
Warn if application...	issuePermission...	true	Security > Global ...
Use realm-qualifi...	useDomainQualifi...	false	Security > Global ...
Use the local sec...	useLocalSecurity...	true	Security > Global ...
Authentication			**Security > Global**
Authentication con...	authConfig	system.KRB5	Security > Global ...
Authentication con...	authContextImplC...	com.ibm.ISecurity...	Security > Global ...
Authentication vali...	authValidationCo...	system.KRB5	Security > Global ...
Enable delegation...	enabledGssCred...	true	Security > Global ...
Kerberos configur...	krb5Config		Security > Global ...
Kerberos keytab file	krb5Keytab		Security > Global ...
Kerberos realm n...	krb5Realm		Security > Global ...
Kerberos service ...	krb5Spn	WAS/${HOST}	Security > Global ...
Simple authentica...	simpleAuthConfig	system.KRB5	Security > Global ...
Trim Kerberos rea...	trimUserName	true	Security > Global ...

Using Color Instead of a Bold Font

Wait a minute, since you're using HTML to make the section rows bold, why don't you just use HTML to add color instead? Listing 21-5 shows the changes made to SecConfigReport_04.py to do just this.

Listing 21-5. SecConfigReport_04.py—HTML Coloring

```
33| html = '<html><font bgcolor=blue color=white>%s</font>'
34| if value :
35| if table.getValueAt( row, 0 ).startswith( '_' ) :
36| if col == 0 :
37| value = value[ 1: ]
38| comp.setText( html % value )
```

This should solve the problem, right? Well, not quite. Figure 21-10 shows that for empty cells the HTML coloring has no effect.

SecConfigReport

Security Settings			
Active authenticati...	activeAuthMechan...	LTPA_1	Security > Global ...
User account repo...	activeUserRegistry	WIMUserRegistry_1	Security > Global ...
Allow basic authe...	allowBasicAuth	true	Security > Global ...
Application security	appEnabled	false	Security > Global ...
Authentication cac...	cacheTimeout	600 seconds	Security > Global ...
Default SSL settin...	defaultSSLSettings	SSLConfig_1	Security > SSL ce...
Dynamically updat...	dynamicallyUpdat...	true	Security > SSL ce...
Administrative sec...	enabled	true	Security > Global ...
Restrict access to ...	enforceFineGrain...	false	Security > Global ...
Java 2 security	enforceJava2Sec...	false	Security > Global ...
Warn if application...	issuePermission...	true	Security > Global ...
Use realm-qualifi...	useDomainQualifi...	false	Security > Global ...
Use the local sec...	useLocalSecurity...	true	Security > Global ...
Authentication			Security > Global
Authentication con...	authConfig	system.KRB5	Security > Global ...
Authentication con...	authContextImplC...	com.ibm.ISecurity...	Security > Global ...
Authentication vali...	authValidationCo...	system.KRB5	Security > Global ...
Enable delegation...	enabledGssCred...	true	Security > Global ...
Kerberos configur...	krb5Config		Security > Global ...
Kerberos keytab file	krb5Keytab		Security > Global ...
Kerberos realm n...	krb5Realm		Security > Global ...
Kerberos service ...	krb5Spn	WAS/${HOST}	Security > Global ...
Simple authentica...	simpleAuthConfig	system.KRB5	Security > Global ...
Trim Kerberos rea...	trimUserName	true	Security > Global ...

You can solve this problem by remembering that the JLabel component that's used by the renderer also has color properties. Don't forget that the component is reused for every cell of the same type, so if you are going to change the cell color for section row cells, you also have to do so for every other cell in the table. Listing 21-6 shows the cell renderer class from SecConfigReport_05.py.

Listing 21-6. SecConfigReport_05.py—Revised Cell Renderer

```
20|class reportRenderer( DefaultTableCellRenderer ) :
21| def __init__( self ) :
22| self.bg = self.fg = None
23| def getTableCellRendererComponent(
24| self,
25| table, # JTable - table containing value
26| value, # Object - value being rendered
```

```
27| isSelected, # boolean - Is value selected?
28| hasFocus, # boolean - Does this cell have focus?
29| row, # int - Row # ( 0 .. N-1 )
30| col # int - Col # ( 0 .. N-1 )
31| ) :
32| DTCR = DefaultTableCellRenderer
33| comp = DTCR.getTableCellRendererComponent(
34| self, table, value, isSelected, hasFocus, row, col
35| )
36| if self.bg == self.fg :
37| self.bg = comp.getBackground()
38| self.fg = comp.getForeground()
39| if value :
40| if table.getValueAt( row, 0 ).startswith( '_' ) :
41| comp.setBackground( Color.blue )
42| comp.setForeground( Color.white )
43| if col == 0 :
44| value = value[ 1: ]
45| comp.setText( '<html><b>%s</b>' % value )
46| else :
47| comp.setBackground( self.bg )
48| comp.setForeground( self.fg )
49| return comp
```

What does this do for the application output? Figure 21-11 shows that this significantly improves the application’s output.

SecConfigReport

Security Settings			
Active authenticati...	activeAuthMechan...	LTPA_1	Security > Global ...
User account repo...	activeUserRegistry	WIMUserRegistry_1	Security > Global ...
Allow basic authe...	allowBasicAuth	true	Security > Global ...
Application security	appEnabled	false	Security > Global ...
Authentication cac...	cacheTimeout	600 seconds	Security > Global ...
Default SSL settin...	defaultSSLSettings	SSLConfig_1	Security > SSL ce...
Dynamically updat...	dynamicallyUpdat...	true	Security > SSL ce...
Administrative sec...	enabled	true	Security > Global ...
Restrict access to ...	enforceFineGrain...	false	Security > Global ...
Java 2 security	enforceJava2Sec...	false	Security > Global ...
Warn if application...	issuePermission...	true	Security > Global ...
Use realm-qualifi...	useDomainQualifi...	false	Security > Global ...
Use the local sec...	useLocalSecurity...	true	Security > Global ...
Authentication			**Security > Global**
Authentication con...	authConfig	system.KRB5	Security > Global ...
Authentication con...	authContextImplC...	com.ibm.ISecurity...	Security > Global ...
Authentication vali...	authValidationCo...	system.KRB5	Security > Global ...
Enable delegation...	enabledGssCred...	true	Security > Global ...
Kerberos configur...	krb5Config		Security > Global ...
Kerberos keytab file	krb5Keytab		Security > Global ...
Kerberos realm n...	krb5Realm		Security > Global ...
Kerberos service ...	krb5Spn	WAS/${HOST}	Security > Global ...
Simple authentica...	simpleAuthConfig	system.KRB5	Security > Global ...
Trim Kerberos rea...	trimUserName	true	Security > Global ...

Figure 21-11. *Output of SecConfigReport_05.py*

Adjusting Column Widths

If you take another look at any of the preceding figures that show the output from any of the scripts, you should notice that even though all of the tables are within scroll panes, only vertical scrollbars are visible. Why do you think that is? Taking a closer look at the tables should give you a hint. The width of the table is fixed, and by default, each column is allocated the same amount of horizontal space. When a value can't be displayed in the available column space, the value is truncated and ellipses appear.

Unfortunately, the table doesn't have any column headings, so you can't adjust the width of individual columns. You can only maximize the application and hope that the screen on which the application is being displayed is wide enough

to allow all of the available information. This isn't a good choice for developers. What you need to do is figure how to deal with this and make some changes to the application so that the users can see all of the information that exists.

In Chapter 20, you learned that you can use the renderer to deal with cells where the data was too wide for the available space. Let's take a closer look at the data that exists in each column to see if you can do the same kind of thing with this application. You can write a quick script to tell you about the maximum number of characters in each column. Note that the last column has a number of entries that contain the arrow, or greater than, symbol >. While you're finding the width of each column, you can also count the maximum number of arrows in each column. Listing 21-7 shows the processReport routine from the columnInfo1.py script; you can find it in the code\Chap_21 directory. **Listing 21-7.** The processReport Routine from the columnInfo1.py Script

```
1|def columnInfo1() :
2| widths = [ 0 ] * 5
3| arrows = [ 0 ] * 5
4| report = AdminTask.generateSecConfigReport()
5| for line in report.splitlines()[ 2: ] :
6| col = 0
7| for cell in line.split( ';' ) :
8| widths[ col ] = max(
9| len( cell.strip() ),
10| widths[ col ]
11| )
12| arrows[ col ] = max(
13| cell.count( '>' ),
14| arrows[ col ]
15| )
16| col += 1
17| print ' widths:', widths
18| print ' arrows:', arrows
```

Figure 21-12 shows the results of using this script on the three most current versions of WebSphere Application Server.

```
* WebSphere version 7.0
  widths: [70, 56, 131, 128, 0]
  arrows: [0, 0, 0, 4, 0]

* WebSphere version 8.0
  widths: [83, 56, 131, 128, 0]
  arrows: [0, 0, 0, 4, 0]

* WebSphere version 8.5
  widths: [83, 55, 131, 128, 0]
  arrows: [0, 0, 0, 4, 0]
```

Figure 21-12. *Output of the columnInfo1.py script*

If you are going to have the cell renderer process the cell data and possibly use HTML to display the cell contents on multiple lines, any routine that determines the preferred column width should use the same kind of processing to more accurately determine the widest line in the column cells. Looking at the results in Figure 21-12, you'll want to take a closer look at the contents of third and fourth columns to reduce the width requirements for these columns.

Column Widths and Row Heights

The next iteration of the script, SecConfigReport_06.py, adds a routine to process the data to determine the maximum width of each column. It also sets the height of each row based on the space required to display the data in the available column width. Listings 21-8 and 21-9 show the setColumnWidths() method.

88| def setColumnWidths(self, table) :
89| tcm = table.getColumnModel() # Table Column Model 90| model = table.getModel() # access the table data 91| margin = tcm.getColumnMargin() # gap between columns 92| rows = model.getRowCount() # How many rows exist? 93| cols = tcm.getColumnCount() # How many columns exist? 94| labels = [
95| JLabel(font = plainFont),
96| JLabel(font = boldFont)
97|]
98| print ' Now Min Pre Max'
99| print '---------------+---+---+---+---'

```
100| metrics = [ fmPlain, fmBold ]
101| tWidth = 0 # Table width
102| section = 0 # is this row a section?
103| sections = 0 # Number of sections
104| for i in range( cols ) : # i == column index
105| col = tcm.getColumn( i )
106| idx = col.getModelIndex()
107| cWidth = 0 # Initial column width
108| for row in range( rows ) :
109| v0 = model.getValueAt( row, 0 )
110| if v0.startswith( '_' ) :
111| section = 1
112| sections += 1
113| else :
114| section = 0
115| comp = labels[ section ]
116| fm = metrics[ section ] # FontMetric
117| r = table.getCellRenderer( row, i )
118| v = model.getValueAt( row, idx )
119| if v.startswith( '_' ) :
120| v = v[ 1: ]
121| comp.setText( v )
122| cWidth = max(
123| cWidth,
124| comp.getPreferredSize().width
125| )
```

One thing that might catch your eye when you look at Listing 21-8 is the code on lines 111 and 112. What’s the difference between the section and sections variables? The former is used to indicate when the current row is a section header. So its value will be 0 or 1 to indicate this. If you search for occurrences of this variable, you will see where it is used as an index to both the labels and metrics arrays (lines 115 and 116). The sections variable is used to count the total number of sections processed in the report.

```
126| if cWidth > 0 :
127| col.setMinWidth( 128 + margin )
128| col.setPreferredWidth( 128 + margin ) 129| col.setMaxWidth( cWidth +
margin )
130| print 'Col: %d widths |%3d|%3d|%3d|%d' % ( 131| i,
```

```
132| col.getWidth(),
133| col.getMinWidth(),
134| col.getPreferredWidth(),
135| col.getMaxWidth()
136| )
137| tWidth += col.getPreferredWidth()
138| print '---------------+---+---+---+---'
139| h0 = table.getRowHeight()
140| print 'rowHeight:', h0
141| sections /= cols
142| print '#Sections:', sections
143| for row in range( rows ) :
144| lines = 1
145| for i in range( cols ) :
146| col = tcm.getColumn( i )
147| pre = col.getPreferredWidth()
148| idx = col.getModelIndex()
149| val = model.getValueAt( row, idx )
150| if not i :
151| section = val.startswith( '_' ) 152| fm = metrics[ section ] # FontMetric 153|
lines = max(
154| lines, int(
155| round( fm.stringWidth( val ) / pre ) + 1 156| )
157| )
158| table.setRowHeight( row, lines * h0 )
159| table.setPreferredScrollableViewportSize(
160| Dimension(
161| tWidth,
162| sections * table.getRowHeight()
163| )
164| )
```

Figure 21-13 shows the output of SecConfigReport_06.py. Comparing this image with the one in Figure 21-11, you can see that the most significant difference is the heights of the individual rows.

Figure 21-13. *Output of SecConfigReport_06.py*

Adding a Frame Resize Listener

The next iteration of the script adds code to display the cell data (the width and height). Additionally, a new event handler routine determines how the information is displayed when the frame is resized. Listing 21-10 contains the source for this event handler.

Listing 21-10. The frameResized Method from SecConfigReport_07.py

```
82| def frameResized( self, ce ) :
83| try :
84| table = self.table
85| model = table.getModel() # Access the table data
86| width = table.getParent().getExtentSize().getWidth()
87| pWidth = int( width ) >> 2
88| tcm = table.getColumnModel() # Table Column Model
89| margin = tcm.getColumnMargin() # gap between columns
```

```
90| cols = tcm.getColumnCount()
91| for c in range( cols ) :
92| col = tcm.getColumn( c )
93| w = min( col.getMaxWidth, pWidth )
94| col.setWidth( w )
95| col.setPreferredWidth( w )
96| height = table.getRowHeight()
97| for row in range( model.getRowCount() ) :
98| table.setRowHeight( row, height )
99| table.repaint()
100| except :
101| print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
```

What does this mean as far as the application output is concerned? Figure 21-14 shows the initial, default rendering of the table and Figure 21-15 shows how the text changes when the frame, and therefore the table, are widened.

Security Settings			
Active authentication mechanism	activeAuthMechanism	LTPA_1	Security > Global security > Authentication mechanisms and expiration
User account repository	activeUserRegistry	WIMUserRegistry_1	Security > Global security > User account repository
Allow basic authentication	allowBasicAuth	true	Security > Global security > Allow basic authentication
Application security	appEnabled	false	Security > Global security > Application security
Authentication cache timeout	cacheTimeout	600 seconds	Security > Global security > Authentication mechanisms and expiration > Authentication expiration
Default SSL settings	defaultSSLSettings	SSLConfig_1	Security > SSL certificate and key management > Manage endpoint security configurations
			Security > SSL certificat

Figure 21-14. *Narrow output of SecConfigReport_07.py*

SecConfigReport

Security Settings			
Active authentication mechanism	activeAuthMechanism	LTPA_1	Security > Global security > Authentication mechanisms and expiration
User account repository	activeUserRegistry	WIMUserRegistry_1	Security > Global security > User account repository
Allow basic authentication	allowBasicAuth	true	Security > Global security > Allow basic authentication
Application security	appEnabled	false	Security > Global security > Application security
Authentication cache time out	cacheTimeout	600 seconds	Security > Global security > Authentication mechanisms and expiration > Authentication expiration
Default SSL settings	defaultSSLSettings	SSLConfig_1	Security > SSL certificate and key management > Manage endpoint security configurations
Dynamically update run time when SSL configuration changes occur	dynamicallyUpdateSSLConfig	true	Security > SSL certificate and key management > Dynamically update run time when SSL configuration changes occur

Figure 21-15. *Wider output of SecConfigReport_07.py*

Notice how the cell contents are wrapped (especially in the first and last columns). Figure 21-15 shows what happens to the cell contents when the frame is widened. It is important that you notice how the row heights are adjusted based on how much vertical space is required to display the text.

Fixing the Row Selection Colors

Unfortunately there is a problem with this iteration of the script. If you test it by selecting a row, you'll witness the problem. There is no visual indication that a row has been selected. Fortunately, the changes required to fix this issue are localized to the table cell renderer code. The most significant changes in the reportRenderer class are these:

• The constructor initializes some arrays to hold the background colors for both selected and unselected rows.
• The getTableCellRendererComponent(...) method now:
• Saves the background and foreground colors for selected and unselected rows when they are first encountered.
• Decides which background and foreground color to use based on whether a row is selected and whether it is one of the header rows.

Figure 21-16 shows the sample output of the modified code. It can be found in the SecConfigReport_08.py script file.

SecConfigReport

Security Settings			
Active authentication mechanism	activeAuthMechanism	LTPA_1	Security > Global security > Authentication mechanisms and expiration
User account repository	activeUserRegistry	WIMUserRegistry_1	Security > Global security > User account repository
Allow basic authentication	allowBasicAuth	true	Security > Global security > Allow basic authentication
Application security	appEnabled	false	Security > Global security > Application security
Authentication cache timeout	cacheTimeout	600 seconds	Security > Global security > Authentication mechanisms and expiration > Authentication expiration
Default SSL settings	defaultSSLSettings	SSLConfig_1	Security > SSL certificate and key management > Manage endpoint security configurations
			Security > SSL certificat

Figure 21-16. *Output of SecConfigReport_08.py*

Which Rows Are Visible?

The next iteration of the script, shown in SecConfigReport_09.py, adds a descendent of the ChangeListener class to monitor changes in the viewport (which is a kind of JViewport[3]), which is the parent of the JTable containing the application information.[4] When the user views a different part of the table, the change listener event handler is called and it prints some information about which rows of the table are visible. Listing 21-11 shows the rowFinder class from this iteration of the script.

Listing 21-11. The rowFinder Class from SecConfigReport_09.py

```
88|class rowFinder( ChangeListener ) :
89| def __init__( self, table ) :
90| self.table = table
```

```
91| def stateChanged( self, ce ) :
92| vPort = ce.getSource()
93| table = self.table
94| rect = vPort.getViewRect()
95| first = table.rowAtPoint(
96| Point( 0, rect.y )
97| )
98| last = table.rowAtPoint(
99| Point( 0, rect.y + rect.height - 1 ) 100| )
101| print 'rows: %d..%d isValid: %d' % ( 102| first,
103| last,
104| vPort.isValid()
105| )
```

I was hoping that the result of calling the isValid() method could be used to determine which rows are visible. Unfortunately, the result returned from this method wasn't quite what I expected. When it returns true, you can easily determine which rows are visible using code similar to Listing 21-11. However, the first and last rows that are visible include partial rows. So you can't use the result of this method to easily determine which complete table rows are visible. To do that requires additional computation.

If you look again at the output in Figure 21-17, you'll see that the routine is invoked multiple times. For most of these the result of the isValid() method is false. Eventually it will return a value of true, which is when you can determine which rows are partially visible. If you look back at Figure 21-16, you can see that rows 0 through 6 are fully visible, but only a little bit of row 7 is shown.

[3] See http://docs.oracle.com/javase/7/docs/api/javax/swing/JViewport.html.

[4]You can display the value returned by vPort.getClass(),which tells you that the variable refers to an instance of javax.swing. JViewport.

```
rows: 0..23  isUalid: 0
rows: 0..23  isUalid: 0
rows: 0..7  isUalid: 0
rows: 0..23  isUalid: 0
rows: 0..23  isUalid: 0
rows: 0..7  isUalid: 0
rows: 0..7  isUalid: 0
rows: 0..7  isUalid: 0
rows: 0..7  isUalid: 1
```

Figure 21-17. *Sample output of the rowFinder stateChanged(...) method*

Table Alignment in the Viewport

The previous iteration shows that sometimes you try something and realize that it isn't going to work out as well as you had hoped. That's one of the really wonderful things about software. Unlike something like woodworking, software is infinitely malleable. You can change it and then change it right back if you don't like the change! In this next iteration, SecConfigReport_10.py, you'll remove the rowFinder class and add the upDownAction class shown in Listing 21-12.

Listing 21-12. Part 1 of the upDownAction Class from SecConfigReport_10.py

```
25|class upDownAction( AbstractAction ) :
26| def __init__( self, table, keyName ) :
27| self.table = table
28| ks = KeyStroke.getKeyStroke( keyName )
29| self.up = keyName.find( 'UP' ) > -1
30| self.action = action = table.getInputMap(
31| JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT
32| ).get( ks )
33| self.original = table.getActionMap().get( action )
34| self.table.getActionMap().put( action, self )
| ...
181| keys = 'UP,DOWN,PAGE_UP,PAGE_DOWN,ctrl END'.split( ',' )
182| for key in keys :
183| upDownAction( table, key )
```

Don't be confused by the indentation in Listing 21-12. The last three lines (181-

183) are not part of the upDownAction class constructor. They are, in fact, from the run(...) method of the SecConfigReport class. However, it is much easier to include them in this listing than it is to create one just for those three lines. Additionally, since these lines are the only reference to this class, it makes sense to show them here to allow for a more complete understanding of how the class is instantiated.

What does this class do? The hint can be found in lines 181-183. It is here that the class is used. Note how the class constructor uses the table instance and the name of a keystroke. In lines 26-34, you can see how the class constructor uses the table to determine the action currently associated with the specified keystroke. This action is saved in an instance variable (called self.original) and replaces it with the action being constructed.

Listing 21-13 shows the remainder of the upDownAction class, specifically the actionPerformed(...) method, which is invoked when a key-related event is generated. When the user presses any of these keys, the newly constructed Action instance is invoked. It uses the original keystroke action, and then adjusts the viewport to better align the table within the available space.

Without this alignment, the top and/or bottom of the viewport may be somewhere in the middle of the row. So, the top of the viewport could be bisecting the row, and the bottom of the viewport might be showing a similar fraction of the row at the bottom. The role of the upDownAction class is to adjust the viewport to align with the bottom of the last visible row when the keystroke used is moving down. When the direction is up, the alignment of the viewport will be with the top row visible row.

Why is this necessary? The simple answer is that when the table rows have variable heights, the default movement actions don’t align the table rows nicely in the viewport. So, this is all related to the fact that the information you want to display doesn’t fit well in the available horizontal space.

Listing 21-13. Part 2 of the upDownAction Class from SecConfigReport_10.py

```
35| def actionPerformed( self, actionEvent ) :
36| table = self.table
37| self.original.actionPerformed( actionEvent )
38| if self.action == 'selectLastRow' :
```

```
39| self.original.actionPerformed( actionEvent )
40| vPort = table.getParent()
41| rect = vPort.getViewRect()
42| row = table.getSelectedRow()
43| if row > -1 :
44| cRect = table.getCellRect( row, 0, 1 )
45| rBot = rect.y + rect.height # Bottom of viewPort
46| cBot = cRect.y + cRect.height # Bottom of cell
47| if rect.y <= cRect.y and rBot >= cBot :
48| return
49| if self.up :
50| first = table.rowAtPoint( Point( 0, rect.y ) )
51| cell = table.getCellRect( first, 0, 1 )
52| diff = rect.y - cell.y
53| else :
54| if row > -1 :
55| cell = table.getCellRect( row, 0, 1 )
56| else :
57| last = table.rowAtPoint(
58| Point(
59| 0,
60| rect.y + rect.height - 1
61| )
62| )
63| cell = table.getCellRect( last, 0, 1 ) 64| bot = rect.y + rect.height
65| end = cell.y + cell.height
66| diff = end - bot
67| point = vPort.getViewPosition()
68| vPort.setViewPosition(
69| Point( point.x, point.y + diff ) 70| )
```

Table Row Filtering

In this next iteration, called SecConfigReport_11.py, you add menu items and row filtering. What’s row filtering? Well, you’ve already seen that the application can identify which rows are section heading rows.

When I was first thinking about this application, I wondered whether there was any easy way to make only specific sections of the report visible. I thought that

maybe I could have each section on a CardLayout[5] (see Chapter 5) or on a JTabbedPane.[6] The problem with using a tabbed pane is that there are too many sections. An application with two dozen tabs wouldn't look very good, would it? If you used a CardLayout, you would need some way to specify which section should be displayed.

That's when I remembered row filtering. Row filtering enables your application to decide which table rows to display. What does that mean for the output? Figure 21-18 shows some sample output of this iteration of the application. The first image shows the initial application display. Here you can see the new Show and Help menu items. The second image shows that the Show menu has three sub-menu items, called Collapse All, Expand All, and Exit. The last image shows the result of selecting Collapse All—the row filtering hides the non-section heading table rows.

[5]See http://docs.oracle.com/javase/7/docs/api/java/awt/CardLayout.html. [6]See http://docs.oracle.com/javase/7/docs/api/javax/swing/JTabbedPane.html.

SecConfigReport

Show Help

Security Settings			
Active authentication mechanism	activeAuthMechanism	LTPA_1	Security > Global security > Authentication mechanisms and expiration
User account repository	activeUserRegistry	WIMUserRegistry_1	Security > Global security > User account repository
Allow basic authentication	allowBasicAuth	true	Security > Global security > Allow basic authentication
Application security	appEnabled	false	Security > Global security > Application security
Authentication cache timeout	cacheTimeout	600 seconds	Security > Global security > Authentication mechanisms and expiration > Authentication expiration
Default SSL settings	defaultSSLSettings	SSLConfig_1	Security > SSL certificate and key management > Manage endpoint security configurations
			Security > SSL certificat

SecConfigReport

Show Help

Collapse all

Expand all

Exit

	activeAuthMechanism	LTPA_1	Security > Global security > Authentication mechanisms and expiration
User account repository	activeUserRegistry	WIMUserRegistry_1	Security > Global security > User account repository
Allow basic authentication	allowBasicAuth	true	Security > Global security > Allow basic authentication
Application security	appEnabled	false	Security > Global security > Application security
Authentication cache timeout	cacheTimeout	600 seconds	Security > Global security > Authentication mechanisms and expiration > Authentication expiration
Default SSL settings	defaultSSLSettings	SSLConfig_1	Security > SSL certificate and key management > Manage endpoint security configurations
			Security > SSL certificat

SecConfigReport

Show Help

Security Settings			
Authentication mechanisms and expiration			Security > Global security > Authentication mechanisms and expiration
User Registry			Security > Global security > User account repository
Authorization configuration			Security > Global security > External authorization providers
Application login configuration			Security > Global security > Java Authentication and Authorization Service
CSI			Security > Global security > RMI/IOP security
SAS			Security > Global security > RMI/IOP security
SSL configuration repertoires			Security > SSL certificate and key management > Manage endpoint security configurations

Figure 21-18. *Output of SecConfigReport_11.py*

Listing 21-14 shows the changes to SecConfigReport_11.py to make this happen. One of the interesting things is how simple the descendent of the abstract RowFilter class[7] is. In fact, as you can see in lines 98-100, the class needs only three lines. However, in order to do this, you need to add two methods to the table model class called reportTable Model—specifically the setVisible() and isVisible() methods shown in lines 109-112. These methods use a new class attribute—the self.visible array—to initialize the constructor on line 104. This array has a Boolean value indicating whether the corresponding row is visible.

Listing 21-14. Row Filtering Changes in SecConfigReport_11.py

```
98|class sectionFilter( RowFilter ) :
99| def include( self, entry ) :
100| return entry.getModel().isVisible( entry.getIdentifier() ) 101|class
reportTableModel( DefaultTableModel ) :
102| def __init__( self, data, headings ) :
103| DefaultTableModel.__init__( self, data, headings ) 104| self.visible = [ 1 ] *
len( data )
| ...
109| def isVisible( self, row ) :
110| return self.visible[ row ]
111| def setVisible( self, row, trueFalse ) :
112| self.visible[ row ] = trueFalse
| ...
189| def collapse( self, event ) :
190| table = self.table
191| model = table.getModel()
192| for row in range( model.getRowCount() ) :
193| model.setVisible(
194| row,
195| model.getValueAt( row, 0 ).startswith( '_' ) 196| )
197| table.getRowSorter().setRowFilter(
198| sectionFilter()
199| )
| ...
202| def expand( self, event ) :
```

```
table = self.table
model = table.getModel()
for row in range( model.getRowCount() ) :
model.setVisible( row, 1 )
table.getRowSorter().setRowFilter(
sectionFilter()
)
```

The initial implementations of the event handler routines invoked by the menu items collapse the sections by setting the non-section rows visible entry to 0 (false) and expand the sections by setting every row's visible entry to 1 (true).

[7]See http://docs.oracle.com/javase/7/docs/api/javax/swing/RowFilter.html.

Finding Text

The next iteration, called SecConfigReport_12.py, adds another menu item so the users can specify text to be found and highlighted. This works in conjunction with row filtering so that the user can collapse all rows, find the text of interest, and make those rows visible as well. Figure 21-19 shows the sample output of this new feature.

Figure 21-19. *Output of SecConfigReport_12.py*

The output shown in Figure 21-19 was produced using the following steps:

1. Choose Show ➤ Collapse All to hide all of the non-section heading rows.
2. Choose Show ➤ Find. Type FIPS in the input field and press the Enter key.
3. Press Ctrl-End to reposition the view to the bottom of the table.[8]

Listings 21-15 and 21-16 show the majority of the code changes required to add this capability to the script. One choice that made this change particularly easy was to add the findText attribute to the reportTableModel class. With this attribute and some getter and setter methods (getFindText(...) and setFindText(...)), the row filter class (sectionFilter) can easily determine what, if any, text needs to be located in the row to ensure that it will be displayed.

Listing 21-15. Part 1 of the Changes Made to SecConfigReport_12.py

[8]Note that a different color scheme indicates when a section row is selected (as you can see on the last row of the table).

```
45|hilightHTML = '<font bgcolor="green" color="yellow">%s</font>' |...
99|class sectionFilter( RowFilter ) :
100| def include( self, entry ) :
101| model = entry.getModel()
102| result = model.isVisible( entry.getIdentifier() ) 103| findText =
model.getFindText()
104| if findText :
105| for col in range( entry.getValueCount() ) : 106| if entry.getStringValue( col
).find( findText ) > 107| result = 1
108| break
109| return result
110|class reportTableModel( DefaultTableModel ) :
111| def __init__( self, data, headings ) :
| ...
114| self.findText = None
| ...
121| def getFindText( self ) :
122| return self.findText
123| def setFindText( self, text ) :
124| self.findText = text
```

The table cell renderer, shown in Listing 21-16 can use the same getter method to see if the text needs to be matched and highlighted in the cell. The hilightHTML variable shown on line 189 refers to a global format string that is used to highlight the specified text using a simple HTML font tag. The variable assignment is on line 45 in Listing 21-15.

Listing 21-16. Part 2 of the Changes Made to SecConfigReport_12.py

```
127|class reportRenderer( DefaultTableCellRenderer ) : | ...
181| result = result.replace(
182| '>', '&gt;'
183| ).replace( '\n', '<br>' )
184| findText = table.getModel().getFindText()
185| if findText :
186| if result.find( findText ) > -1 :
187| result = result.replace(
188| findText,
189| hilightHTML % findText
```

```
190| )
191| value = '<html>' + result.replace( ' ', ' ' ) |...
194|class SecConfigReport_12( java.lang.Runnable ) : |...
232| def Find( self, event ) :
233| result = JOptionPane.showInputDialog( 234| self.frame, #
parentComponent 235| 'Text to be found:' # message text 236| )
237| self.table.getModel().setFindText( result ) 238|
self.table.getRowSorter().setRowFilter( 239| sectionFilter()
240| )
```

This code simply locates instances of the user-specified text and replaces that text with the HTML that highlights the text for the user.

Section Visibility

If you use this iteration for a short time, you will find that being able to collapse and expand all sections is nice, but not quite good enough. Wouldn’t it be great if you could show or hide individual sections simply by double-clicking on the section row?

Figure 21-20 shows a sample image from the latest iteration of the application, SecConfigReport_13.py. In it, you can see that all sections have been collapsed, and the view has been scrolled about half way down the table until the section named Management Scope is visible at the top of the viewport. The image shows the output after finding the rows containing scopeName; three additional rows are now visible.

Figure 21-20. *Output of SecConfigReport_13.py*

Listings 21-16 through 21-20 show the code changes that provide this capability. Most of the changes are to the reportTableModel class in order to add attributes and methods to identify the sections in the table.

Let's start by taking a look at the modified sectionFilter class, which is shown in Listing 21-17. The only changes made here are on lines 101 through 103. The role of this method is to return true (1) when the row specified by the entry argument should be visible. In previous versions of this class, the row would be visible only if the whole section was visible or if the user specified some text that exists in the current row.

Listing 21-17. Modified sectionFilter Class from SecConfigReport_13.py

```
98|class sectionFilter( RowFilter ) :
99| def include( self, entry ) :
100| model = entry.getModel()
101| row = entry.getIdentifier()
```

```
102| section = model.getSectionNumber( row )
103| result = model.isRowVisible( row ) or
| model.isSectionVisible( section )
104| findText = model.getFindText()
105| if findText :
106| for col in range( entry.getValueCount() ) :
107| if entry.getStringValue( col ).find( findText ) > -1 : 108| result = 1
109| break
110| return result
```

Changes to the include(...) method (see line 103) assume that one section is visible. You can see this by collapsing all of the sections (choose Show ➤ Collapse All) and then double-clicking on one of the visible section headings.

Listing 21-18. The reportTableModel Class from SecConfigReport_13.py

```
111|class reportTableModel( DefaultTableModel ) :
112| def __init__( self, data, headings ) :
113| DefaultTableModel.__init__( self, data, headings )
114| L = len( data )
115| self.visible = [ 1 ] * L
116| self.sectionNumber = [ 0 ] * L
117| self.sections = 0
118| section = -1
119| for i in range( L ) :
120| row = data[ i ]
121| if row[ 0 ].startswith( '_' ) :
122| self.sections += 1
123| section += 1
124| self.sectionNumber[ i ] = section
125| self.sectionVisible = [ 1 ] * self.sections
126| self.findText = None
127| def getColumnClass( self, col ) :
128| return String
129| def getFindText( self ) :
130| return self.findText
131| def getSectionCount( self ) :
132| return self.sections
133| def getSectionNumber( self, row ) :
134| try :
```

```
135| result = self.sectionNumber[ row ] 136| except :
137| result = -1
138| return result
139| def isCellEditable( self, row, col ) :
140| return 0
141| def isSectionVisible( self, sectionNum ) : 142| try :
143| result = self.sectionVisible[ sectionNum ] 144| except :
145| result = 0
146| return result
147| def isRowVisible( self, row ) :
148| return self.visible[ row ]
149| def setFindText( self, text ) :
150| self.findText = text
151| def setSectionVisible( self, sectionNum, trueFalse ) : 152|
self.sectionVisible[ sectionNum ] = trueFalse 153| def setRowVisible( self, row,
trueFalse ) : 154| self.visible[ row ] = trueFalse
```

The changes to the sectionFilter class are not the only ones you need to make; you also have to make some significant changes to the reportTableModel class. This revised class is shown in Listing 21-18. You should be able to see that most of the changes to this class deal with identifying sections and being able to determine the section number for each row in the data.

You also need to add a mouse listener event handler in the application class (SecConfigReport) to detect when the user double-clicks on the table. The mouseClicked event handler method shown in Listing 21-19 is added to each row of the table. It is important to note, however, that it only changes the visibility of a section by checking that the row that was double-clicked is a section heading (see line 241).

Listing 21-19. The clicker(...) Method in SecConfigReport_13.py

```
217|class SecConfigReport_13( java.lang.Runnable ) :
| ...
234| def clicker( self, event ) :
235| if event.getClickCount() == 2 :
236| table = event.getSource()
237| model = table.getModel()
238| view = row = table.getSelectedRow()
```

```
239| if view > -1 :
240| row = table.convertRowIndexToModel( view )
241| if model.getValueAt( row, 0 ).startswith( '_' ) :
242| sNum = model.getSectionNumber( row )
243| model.setSectionVisible(
244| sNum,
245| not model.isSectionVisible( sNum )
246| )
247| else :
248| sNum = -1
249| else :
250| sNum = -1
251| table.getRowSorter().setRowFilter( 252| sectionFilter()
253| )
```

The other significant changes are made to the Find(...) method so that it takes the table model section attributes and methods into account. The modified Find(...) method is shown in Listing 21-20.

Listing 21-20. Modified Find(...) Method from SecConfigReport_13.py

```
279| def Find( self, event ) :
280| table = self.table
281| cols = table.getColumnModel().getColumnCount()
282| model = table.getModel()
283| result = JOptionPane.showInputDialog(
284| self.frame, # parentComponent
285| 'Text to be found:' # message text
286| )
287| model.setFindText( result )
288| for row in range( model.getRowCount() ) :
289| visible = model.getValueAt(
290| row,
291| 0
292| ).startswith( '_' )
293| if result and not visible :
294| for col in range( cols ) :
295| val = model.getValueAt( row, col )
296| if val.find( result ) > -1 :
297| visible = 1
```

```
298| break
299| model.setRowVisible( row, visible )
300| table.getRowSorter().setRowFilter(
301| sectionFilter()
302| )
```

Does It Work?

One thing that you should have learned by this point is that you always need to test your applications to verify that they act as you expect. To see if the double-click event handler is doing its job, you can use the Show ➤ Collapse All menu selections, and then double-click the visible section heading rows to see if the associated rows become visible. This test appears to validate the expectations. What else can you test? Does it work correctly when you double-click a section heading row after expanding the rows? See Figure 21-21.

Figure 21-21. *Testing SecConfigReport_13.py*

Unfortunately, no, it doesn’t. What’s going on? Why doesn’t it work correctly? Take another look at the include(...) method in the sectionFilter class in Listing 21-17. Under what conditions will a row be visible? Line 103 shows that in order for a row to be hidden, both isRowVisible(...) and isSectionVisible(...) must return false.

What happens to the mouseClicked event handler method when a section heading row is the target of the event? Which of these model properties is affected? Only the sectionVisible property is affected, so the visible property of each row in the section is unaffected. That’s the problem right there. To fix this, the event handler has to change the visible property for each row in the section.

Listing 21-21 shows the modified clicker(...) method event handler. Note how the target row is verified as a section header row. Its new visibility is determined (line 243), set (line 244), and then used to assign the visibility of each row within the section (lines 245 through 249).

Listing 21-21. Modified clicker() Method from SecConfigReport_14.py

```
234| def clicker( self, event ) :
235| if event.getClickCount() == 2 :
236| table = event.getSource()
237| model = table.getModel()
238| view = row = table.getSelectedRow()
239| if view > -1 :
240| row = table.convertRowIndexToModel( view )
241| if model.getValueAt( row, 0 ).startswith( '_' ) :
242| sNum = model.getSectionNumber( row )
243| vis = not model.isSectionVisible( sNum )
244| model.setSectionVisible( sNum, vis )
245| for row in range( row + 1, model.getRowCount() ) : 246| if
model.getSectionNumber( row ) == sNum : 247| model.setRowVisible( row, vis
) 248| else :
249| break
250| else :
251| sNum = -1
252| else :
253| sNum = -1
254| table.getRowSorter().setRowFilter(
```

```
255| sectionFilter()
256| )
```

Please keep in mind that the event handler code should be done quickly. You don't want the routine to delay the update. If you test this version of the application, you'll see that the event handler doesn't take too long to complete.

Progress Indicator

In relation to the possibility of a delay in the event handler, I wondered also about the delay that was occurring when the script begins to execute. A bit of testing shows that there is a non-trivial amount of time required to execute the AdminTask.generateSecConfigReport() method. See for yourself. Start an interactive wsadmin session[9] and, when the command prompt is displayed, execute the generateSecConfigReport() method.[10] Take note of how long it takes to complete.

Based on this improved understanding of what was occurring, I tried a few things to display some kind of progress indicator to show the user that something was happening. Unfortunately, I didn't have much luck. My attempt to use a SwingWorker task to perform the AdminTask method call and display an indeterminate progress bar in a dialog box didn't work for some reason. I decided to use a different approach.

Listing 21-22 shows the changes made to this next iteration of the script. As you can see in lines 66-70, a trivial SwingWorker class performs the call to the AdminTask method on a background thread and makes the result available via a simple getter method.

Listing 21-22. Modified run(...) Method from SecConfigReport_15.py

```
66|class reportTask( SwingWorker ) :
67| def doInBackground( self ) :
68| self.results = AdminTask.generateSecConfigReport( | ).splitlines()[ 2: ]
69| def getResults( self ) :
70| return self.results
|...
```

[9]wsadmin -conntype none -lang jython [10]text =

AdminTask.generateSecConfigReport()

```
229|class SecConfigReport_15( java.lang.Runnable ) :
|...
364| def run( self ) :
365| try :
366| task = reportTask()
367| task.execute()
368| chars = r'-\|/'
369| char = 0
370| while not task.isDone() :
371| print '\b%s\b' % chars[ char ],
372| sleep( 0.25 )
373| char = ( char + 1 ) % len( chars )
374| print '\b \b',
375| info = task.getResults()
376| except :
377| print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
378| sys.exit()
|...
```

Lines 366 and 367 show how this task instance is created and executed. You need to save a reference to the task instance object in order to call its getResults() method once the task is complete.[11] Lines 368-373 display a trivial character-oriented indicator to show that the AdminTask method call has not yet completed. Once the task is complete, its results are retrieved, and the application can get on with its purpose.

Is this application perfect? Absolutely not. As indicated earlier, it is merely a proof of concept (PoC). If you wanted to enhance it and use it in a more permanent arrangement, you could base those decisions on what you can see and do with this script. You could also use it to test possible enhancements or improvements.

Summary

The purpose of this chapter is twofold. First it is intended to show how easily you can turn a report into an interactive graphical user application. It is also intended to show that by taking small steps you can more easily verify that each

iteration of the script works as it should. If it isn't, you can more easily determine the source of the problem and correct it.wsadmin

[11]Note the use of the isDone() method on line 370, which is used to determination when the background task is complete. **Chapter 22**

WASports: A WebSphere Port Application

One of the first topics that grabbed my imagination when I started thinking about graphical wsadmin scripts was the possibility of displaying and managing all of the TCP/IP port numbers being used by a WebSphere Application Server cell. Trying to use the administration console to view the port numbers being used by all of the servers in a cell can be frustrating and tedious. I wanted an iterative graphical application that can be used to quickly and easily understand which application servers exist in the cell, and use a tree structure to show the hierarchical relationship between these servers. Additionally I wanted to be able to show the port numbers being used by each of these servers. This chapter shows how to build this application using the same type of iterative approach used previously.

Using the Administration Console

What does it take to use the Administration console to view the port numbers currently being used by a managed application server? The following steps are needed to view and change just one TCP/IP port number for an existing application server using the administrative console.

1. If the Deployment Manager is active, skip to Step 3.
2. Use the startManager command to start the Deployment Manager.

3. Use a browser to access the Administration console associated with the Deployment Manager. Enter the appropriate username and password to view the desired information.

4. Select and expand the Servers section in the left frame.
5. Select and expand the Server Types section.
6. Select the WebSphere Application Servers link.
7. Select the appropriate server link (such as server1).
8. Expand the Ports section under the Communications heading.

9. Use the Details button or the Ports link to show a table of the named endpoints and the associated port number values used by the specified application server.

10. Select the desired endpoint name to be modified, such as BOOTSTRAP_ADDRESS.
11. Modify the Port input field to identify the new port number value.
12. Select the Apply or OK button.
13. Select the Save link to update the master configuration.

14. After all the required changes have been made, stop and restart the server to use the modified port numbers.
Figure 22-1 shows a simple example of a table of the port numbers being used by a single application server. This corresponds to the kind of table that is the result of performing Step 8.

⊟ Ports

Port Name	Port
BOOTSTRAP_ADDRESS	2810
SOAP_CONNECTOR_ADDRESS	8881
ORB_LISTENER_ADDRESS	9102
SAS_SSL_SERVERAUTH_LISTENER_ADDRESS	9409
CSIV2_SSL_SERVERAUTH_LISTENER_ADDRESS	9408
CSIV2_SSL_MUTUALAUTH_LISTENER_ADDRESS	9407
WC_adminhost	9062
WC_defaulthost	9081
DCS_UNICAST_ADDRESS	9354
WC_adminhost_secure	9045
WC_defaulthost_secure	9444
SIP_DEFAULTHOST	5063
SIP_DEFAULTHOST_SECURE	5062
SIB_ENDPOINT_ADDRESS	7278
SIB_ENDPOINT_SECURE_ADDRESS	7287
SIB_MQ_ENDPOINT_ADDRESS	5559
SIB_MQ_ENDPOINT_SECURE_ADDRESS	5579
IPC_CONNECTOR_ADDRESS	9634

Figure 22-1. *Sample list of application server port settings*

If you view and modify the port numbers being used by the Deployment Manager or the node agents, you need to start by selecting and expanding the System Administration section instead of performing Steps 4 and 5. All this is used to show the number of steps needed to view and modify the port numbers being used by an application server using the administration console.

The AdminTask.listServerPorts( ) Method

Using the WAShelp.py script from Chapter 20, you can search for any scripting object methods that identify the ports being used by a server. Figure 22-2 shows what happens when you search for AdminTask methods containing the word "Ports" anywhere in the method name.

Figure 22-2. *AdminTask methods with "Ports" in the name*

The listServerPorts(...) method looks really promising. Let's see what it produces for the dmgr server.[1] The result is a string list identifying the port numbers configured for the specified application server. Unfortunately, the result of calling this method isn't as useful as you might hope it would be. Listing 22-1 shows some sample output from this method. Imagine the processing required to use this information. Few people would think the processing required to parse this output worth the effort. This is especially true since the information is more easily available using other techniques.

Listing 22-1. Sample AdminTask.listServerPorts(...) Output

```
wsadmin>print AdminTask.listServerPorts( 'dmgr' )
[[IPC_CONNECTOR_ADDRESS [[[host ${LOCALHOST_NAME}] [node
RAGibson
-W520CellManager01] [server dmgr] [port 9632] ]]] ]
[[CSIV2_SSL_SERVERAUTH_LISTENER_ADDRESS [[[host RAGibson-
W520.ral
eigh.ibm.com] [node RAGibson-W520CellManager01] [server dmgr] [po
rt 9403] ]]] ]
[[WC_adminhost [[[host *] [node RAGibson-W520CellManager01] [serv
er dmgr] [port 9060] ]]] ]
[[DataPowerMgr_inbound_secure [[[host *] [node RAGibson-W520CellM
anager01] [server dmgr] [port 5555] ]]] ]
[[DCS_UNICAST_ADDRESS [[[host *] [node RAGibson-
W520CellManager01
] [server dmgr] [port 9352] ]]] ]
[[BOOTSTRAP_ADDRESS [[[host RAGibson-W520.raleigh.ibm.com] [node
RAGibson-W520CellManager01] [server dmgr] [port 9809] ]]] ]
[[SAS_SSL_SERVERAUTH_LISTENER_ADDRESS [[[host RAGibson-
W520.ralei
gh.ibm.com] [node RAGibson-W520CellManager01] [server dmgr] [port

9401] ]]] ]
[[SOAP_CONNECTOR_ADDRESS [[[host RAGibson-W520.raleigh.ibm.com]
[ node RAGibson-W520CellManager01] [server dmgr] [port 8879] ]]] ]
[[CELL_DISCOVERY_ADDRESS [[[host RAGibson-W520.raleigh.ibm.com] [
node RAGibson-W520CellManager01] [server dmgr] [port 7277] ]]] ]
[[ORB_LISTENER_ADDRESS [[[host RAGibson-W520.raleigh.ibm.com] [no
de RAGibson-W520CellManager01] [server dmgr] [port 9100] ]]] ]
[[CSIV2_SSL_MUTUALAUTH_LISTENER_ADDRESS [[[host RAGibson-
W520.ral eigh.ibm.com] [node RAGibson-W520CellManager01] [server dmgr]
[po rt 9402] ]]] ]
[[WC_adminhost_secure [[[host *] [node RAGibson-W520CellManager01 ]
[server dmgr] [port 9043] ]]] ]
wsadmin>
```

[1]Note that if the server name is not unique, a qualifying -nodename parameter must be specified.

The AdminTask.reportConfiguredPorts( ) Method

The other interesting AdminTask method listed by the WAShelp script is the reportConfiguredPorts(...) method. Figure 22-3 shows the help text for this method.

Figure 22-3. *Help text for AdminTask.reportConfiguredPorts(...) method*

The description of this method is quite promising. The real question though is what kind of output is generated by this method? Listing 22-2 shows the initial output generated by a call to the AdminTask.reportConfiguredPorts(...) method.

Listing 22-2. Initial Output of reportConfiguredPorts(...) Method Call

```
wsadmin>print AdminTask.reportConfiguredPorts() Ports configured in cell
RAGibson-W520Cell01

Node RAGibson-W520CellManager01 / Server dmgr
RAGibson-W520.raleigh.ibm.com:7277 CELL_DISCOVERY_ADDRESS
RAGibson-W520.raleigh.ibm.com:9809 BOOTSTRAP_ADDRESS
${LOCALHOST_NAME}:9632 IPC_CONNECTOR_ADDRESS
RAGibson-W520.raleigh.ibm.com:8879 SOAP_CONNECTOR_ADDRESS
RAGibson-W520.raleigh.ibm.com:9100 ORB_LISTENER_ADDRESS
RAGibson-W520.raleigh.ibm.com:9401 SAS_SSL_SERVERAUTH_LIST...
RAGibson-W520.raleigh.ibm.com:9402 CSIV2_SSL_MUTUALAUTH_LI...
RAGibson-W520.raleigh.ibm.com:9403 CSIV2_SSL_SERVERAUTH_LI...
*:9060 WC_adminhost
*:9043 WC_adminhost_secure
```

*:9352 DCS_UNICAST_ADDRESS
*:5555 DataPowerMgr_inbound_secure

...
This certainly seems to be much easier to read and understand from the human perspective. However, some challenges remain if you intend to process this kind of result in your scripts.

Using AdminConfig Methods

In Chapter 11, you saw how easy it was to produce a tree hierarchy representing the cell.[2] Let's see what it takes to produce information about the nodes, servers, and their configured ports using a non-graphical script. Listing 22-3 shows most of the main routine needed to do just that.[3]

Listing 22-3. ListPorts Routine from the ListPorts.py Script[4]

```
42|def ListPorts() :
43| gAV = getAttributeValue # For line shortening purposes
44| names = {}
45| nodes = 0
46| for node in AdminConfig.list( 'Node' ).splitlines() :
47| nodes += 1
48| names[ 'nodeName' ] = gAV( node, 'name' )
49| names[ 'profName' ] = profileName( node )
50| SEs = AdminConfig.list( 'ServerEntry', node )
51| servers = 0
52| for se in SEs.splitlines() :
53| servers += 1
54| names[ 'servName' ] = gAV( se, 'serverName' ) 55| names[ 'hosts' ] = ', '.join(
getHostnames( se ) ) 56| print formatString % names
57| data = []
58| NEPs = AdminConfig.list( 'NamedEndPoint', se ) 59| for nep in
NEPs.splitlines() :
60| name = gAV( nep, 'endPointName' )
61| epId = gAV( nep, 'endPoint' )
62| port = gAV( epId, 'port' )
63| data.append( ( port, name ) )
64| data.sort( lambda a, b : cmp( a[ 1 ], b[ 1 ] ) ) 65| for port, name in data :
```

```
66| print '%5d | %s' % ( port, name )
67| print
```

[2] For example, see the code\Chap_11\Tree4.py sample script.

[3]The complete script can be found in the code\Chap_22\ListPorts.py file.

[4]Remember that many of the scripts listed in the book are written to fit within the available horizontal space.

What does the output of this script look like? Listing 22-4 shows the initial portion of the output generated when this script was executed using a WebSphere Application Server V 7.0 environment on my local machine.

Listing 22-4. Sample Output of the ListPorts.py Script

```
Profile name: Dmgr01
Host name(s): RAGibson-W520.raleigh.ibm.com Node name: RAGibson-
W520CellManager01

Server name: dmgr

Port | EndPoint Name
------+-------------
9809 | BOOTSTRAP_ADDRESS
7277 | CELL_DISCOVERY_ADDRESS
9402 | CSIV2_SSL_MUTUALAUTH_LISTENER_ADDRESS
9403 | CSIV2_SSL_SERVERAUTH_LISTENER_ADDRESS
9352 | DCS_UNICAST_ADDRESS
5555 | DataPowerMgr_inbound_secure
9632 | IPC_CONNECTOR_ADDRESS
9100 | ORB_LISTENER_ADDRESS
9401 | SAS_SSL_SERVERAUTH_LISTENER_ADDRESS
8879 | SOAP_CONNECTOR_ADDRESS
9060 | WC_adminhost
9043 | WC_adminhost_secure
```

The output produced by this simple script has some nice properties including the ease of reading and understanding the information. It also corresponds closely with the way in which information is displayed on the Administration console, as shown in Figure 22-1.

Step 0: Creating a WASports Application

Let’s start by creating a simple empty frame application. I choose to call this step 0 because using an existing script template is so trivial. The code in Listing 22-5 should be very familiar to you by now, so I won’t bother to elaborate. The only part that might not be familiar to you is in lines 18 and 19, where a JDesktopPane instance is created and used as the frame content pane. This is discussed in more detail in Chapter 19.

Listing 22-5. WASports Class from the WASports_00.py Script

```
6|class WASports_00( java.lang.Runnable ) :
7| def run( self ) :
8| screenSize = Toolkit.getDefaultToolkit().getScreenSize()
9| w = screenSize.width >> 1 # Use 1/2 screen width 10| h = screenSize.height
>> 1 # and 1/2 screen height 11| x = ( screenSize.width - w ) >> 1 # Top left
corner 12| y = ( screenSize.height - h ) >> 1
13| frame = self.frame = JFrame(
14| 'WASports_00',
15| bounds = ( x, y, w, h ),
16| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 17| )
18| desktop = JDesktopPane()
19| frame.setContentPane( desktop )
20| frame.setVisible( 1 )
```

Step 1: Adding an Empty Internal Frame

Next you need to add a little code to create an empty internal frame to the application desktop. Listing 22-6 shows how this is done in the next iteration of the WASports script (in WASports_01.py).

Listing 22-6. Defining and Using an InternalFrame Class

```
9|class InternalFrame( JInternalFrame ) :
10| def __init__( self, title, size, location, closable = 0 ) : 11|
JInternalFrame.__init__(
12| self,
13| title,
14| resizable = 1,
15| closable = closable,
```

```
16| maximizable = 1,
17| iconifiable = 1,
18| size = size
19| )
20| self.setLocation( location )
21| self.setVisible( 1 )
22|class WASports_01( java.lang.Runnable ) : 23| def run( self ) :

| ...
34| desktop = JDesktopPane()
35| internal = InternalFrame(
36| 'InternalFrame',
37| size = Dimension( w >> 1, h >> 1 ),
38| location = Point( 5, 5 )
39| )
40| desktop.add( internal )
41| frame.setContentPane( desktop )
42| frame.setVisible( 1 )
```

Step 2: Adding an Empty JSplitPane to the Internal Frame

Next you need to add an empty split pane to the internal frame. Listing 22-7 shows how easily this can be done. The code from this listing is from the WASports_02.py script file. The statement in line 30 is used to position the divider in the middle of the internal frame.

Listing 22-7. JSplitPane Code Added to WASports_02.py

```
11|class InternalFrame( JInternalFrame ) :
12| def __init__( self, title, size, location, closable = 0 ) : | ...
22| self.setLocation( location )
23| pane = self.add(
24| JSplitPane(
25| JSplitPane.HORIZONTAL_SPLIT,
26| JLabel( 'Left' ),
27| JLabel( 'Right' )
28| )
29| )
30| pane.setDividerLocation( size.width >> 1 )
```

```
31| self.setVisible( 1 )
```

The JLabel instances in this example are simple placeholders for other kinds of Swing components.

Step 3: Adding a Cell Hierarchy Tree to the JSplitPane

Listing 22-8 shows that very little new code is needed in the WASports_03.py script file in order to create a cell hierarchy tree on the left portion of the split pane. The cellTree(...) method (lines 40-54) contains code very similar to Listing 22-3.

Listing 22-8. JTree Code Added to WASports_03.py

```
15|class InternalFrame( JInternalFrame ) :
16| def __init__( self, title, size, location, closable = 0 ) : | ...
26| self.setLocation( location )
27| tree = self.cellTree()
28| tree.getSelectionModel().setSelectionMode(
29| TreeSelectionModel.SINGLE_TREE_SELECTION
30| )
31| pane = self.add(
32| JSplitPane(
33| JSplitPane.HORIZONTAL_SPLIT,
34| JScrollPane( tree ),
35| JLabel( 'Right' )
36| )
37| )
38| pane.setDividerLocation( size.width >> 1 )
39| self.setVisible( 1 )
40| def cellTree( self ) :
41| cell = AdminConfig.list( 'Cell' )
42| root = DefaultMutableTreeNode( self.getName( cell ) )
43| for node in AdminConfig.list( 'Node' ).splitlines() :
44| here = DefaultMutableTreeNode(
45| self.getName( node )
46| )
47| servers = AdminConfig.list( 'Server', node )
48| for server in servers.splitlines() :
```

```
49| leaf = DefaultMutableTreeNode(
50| self.getName( server )
51| )
52| here.add( leaf )
53| root.add( here )
54| return JTree( root )
55| def getName( self, configId ) :
56| return AdminConfig.showAttribute( configId, 'name' )
```

It’s probably about time to see some sample output of this iteration of the WASports script. Figure 22-4 shows that the WASports application is starting to take shape. The application frame and internal frame can be moved, resized, and so on, as the user desires. It also limits the number of tree nodes that may be selected at one time.

Figure 22-4. *Sample output of the WASports_03.py script*

Step 4: Updating the Right Pane

The next step is to update the other side of the split pane based on the user selections of the tree nodes. This requires a bit more work. In fact, if you simply look at the “lines of code” (ignoring comments and blank lines), this version of the WASports script is about half as large as the previous one. However, it still isn’t too much of a difference because it only requires about 125 lines of code to provide this kind of functionality. What “big changes” are needed to do this?

Listing 22-9 shows the cellTSL class that is used by the script to react to tree selection events.

Listing 22-9. The cellTSL Class from WASports_04.py[5]

```
16|class cellTSL( TreeSelectionListener ) :
17| def __init__( self, tree, pane, name2cfgId ) :
18| self.tree = tree
19| self.pane = pane
20| self.name2cfgId = name2cfgId
21| def valueChanged( self, tse ) :
22| format = (
23| '<html>  node: %s<br/>' +
24| 'isLeaf: %s<br/>parent: %s'
25| )
26| pane = self.pane
27| node = self.tree.getLastSelectedPathComponent()
28| if node :
29| text = format % (
30| node,
31| [ 'No', 'Yes' ][ node.isLeaf() ],
32| node.getParent()
33| )
34| if node.isLeaf() :
35| key = ( str( node.getParent() ), str( node ) ) 36| else :
37| key = node.toString()
38| if self.name2cfgId.has_key( key ) :
39| text += '<br/><br/>%s' % self.name2cfgId[ key ] 40| else :
41| text += '<br/><br/> key missing: %s' % key 42| else :
43| text = 'Nothing selected'
44| pane.setText( text )
```

[5]Where TSL is an acronym for TreeSelectionListener.

When a cellTSL object is instantiated, the caller must provide references to:
• The JTree instance being monitored
• The pane being updated
• A dictionary containing information to be used to update the pane

The first two parameters are likely to be obvious, but the last is less likely to be so. This dictionary is used in lines 38 and 39 simply to demonstrate how the selection can be used to retrieve the associated configuration ID from the dictionary. For this example, this configuration ID is part of the information that is displayed in the right pane when a tree node is selected.

Listing 22-10. Tree and Split Pane Creation Code from WASports_04.py

```
| ...
56| self.setLocation( location )
57| tree, self.name2cfgId = self.cellTree()
58| tree.getSelectionModel().setSelectionMode(
59| TreeSelectionModel.SINGLE_TREE_SELECTION
60| )
61| self.status = JLabel( 'Right' )
62| tree.addTreeSelectionListener(
63| cellTSL(
64| tree,
65| self.status,
66| self.name2cfgId
67| )
68| )
69| pane = self.add(
70| JSplitPane(
71| JSplitPane.HORIZONTAL_SPLIT,
72| JScrollPane( tree ),
73| JScrollPane( self.status )
74| )
75| )
```

Listing 22-10 shows the modification to the WASports_04.py script to instantiate the TreeSelectionListener for the cell hierarchy tree. It’s important to note that the constructor call requires a dictionary as the third argument. Listing 22-11 shows the modifications that were made to the cellTree() method in order to build and return this dictionary. It is important to note that the dictionary index is simple for non-leaf tree nodes (the root and all of the nodes corresponding to WebSphere node names). It is only the leaf nodes that use a tuple as an index. Why is this? Because the leaf nodes correspond to the individual application servers in the cell, the names of which do not have to be unique. The server names only have to be unique within a node. The dictionary entries for the

individual servers use a tuple composed of the node name and the server name (see line 93).

Listing 22-11. Modified cellTree() Method

```
78| def cellTree( self ) :
79| cell = AdminConfig.list( 'Cell' )
80| cellName = self.getName( cell )
81| root = DefaultMutableTreeNode( cellName )
82| result = { cellName : cell }
83| for node in AdminConfig.list( 'Node' ).splitlines() :
84| nodeName = self.getName( node )
85| here = DefaultMutableTreeNode(
86| nodeName
87| )
88| result[ nodeName ] = node
89| servers = AdminConfig.list( 'Server', node )
90| for server in servers.splitlines() :
91| name = self.getName( server )
92| leaf = DefaultMutableTreeNode( name )
93| result[ ( nodeName, name ) ] = server
94| here.add( leaf )
95| root.add( here )
96| return JTree( root ), result
```

What does this mean for the application? Figure 22-5 shows how the right pane is updated based on user selections on the tree in the left pane. The second image shows the result of selecting a server. Note how the bottom line of the right pane is the configuration ID for the selected server.

Figure 22-5. *Sample output of the WASports_04.py script*

Step 5: Displaying Cell and Node Details

A reasonable next step is to see what you can do to improve the information displayed in the right panel when the cell and node (non-leaf) tree items are selected. Figure 22-6 has some sample output from the next iteration of this script. It shows the different kinds of information displayed depending on the various types of tree nodes that can be selected. The first shows what is displayed when the cell node is selected, the second shows the slightly different information displayed when a WebSphere node is selected, and the last shows a minimum amount of information when a server entry is selected.

Figure 22-6. *Sample output of the WASports_05.py script*

Utility Routines

At first glance, it might appear simple to determine the values being displayed in the images in Figure 22-6. Unfortunately, it takes a bit more effort and code than you might imagine. For example, Listing 22-12 includes the utility routines needed to obtain some of this information.

Listing 22-12. Part 1 of the WASports_05.py Utility Routines

```
def findScopedTypes( Type, value, scope = None, attr = None ) :
```

```
40| if not attr :
41| attr = 'name'
42| return [
43| x for x in AdminConfig.list( Type, scope ).splitlines()
44| if getAttributeValue( x, attr ) == value
45| ]
46|def getAttributeValue( cfgId, attr ) :
47| return AdminConfig.showAttribute( cfgId, attr ) 48|def getIPaddresses(
hostnames ) :
49| result = []
50| for hostname in hostnames :
51| try :
52| addr = gethostbyname( hostname ) 53| if addr not in result :
54| result.extend( addr.split( ',' ) ) 55| except :
56| pass
57| return result
58|def getHostnames( nodeName, serverName ) :
59| exclude = [
60| '*',
61| 'localhost',
62| '${LOCALHOST_NAME}',
63| '232.133.104.73',
64| 'ff01::1'
65| ]
66| result = []
67| node = findScopedTypes( 'Node', nodeName )[ 0 ] 68| server =
findScopedTypes(
69| 'ServerEntry',
70| serverName,
71| node,
72| 'serverName'
73| )[ 0 ]
74| NEPs = AdminConfig.list( 'NamedEndPoint', server ) 75| for nep in
NEPs.splitlines() :
76| epId = getAttributeValue( nep, 'endPoint' ) 77| host = getAttributeValue(
epId, 'host' ) 78| if host not in exclude :
79| result.append( host )
80| exclude.append( host )
81| return result
```

Table 22-1 describes each of the routines found in Listing 22-12. It is interesting to see how much of these routines use WebSphere specific objects and attribute values.

Table 22-1. *Part 1 of the WASports_05.py Utility Routines, Explained*

Lines	Description
39–45	The findScopedTypes(...) routine is used to return a list of configuration IDs for configuration objects of the specified Type that have a particular attribute value.
46–47	The getAttributeValue(...) routine is simply a technique used to shorten lines that would normally call the Adminconfig.showAttribute(...) method.
48–57	The getIPaddresses(...) routine returns a list IP addresses for the specified hostname.
58–81	The getHostnames(...) routine returns a list of unique hostnames referenced by the endPoint configuration objects on the specified server. **Note:** Since server names are only guaranteed to be unique within a node, the node name also needs to be provided.

Listing 22-13 contains some additional utility routines, all of which have names beginning with “WAS” to indicate how specific they are to WebSphere Application Server product information. In fact, most of them use the WebSphere product variables in order to determine the value to be returned.

Listing 22-13. Part 2 of the WASports_05.py Utility Routines

```
82|def WASversion( id ) :
83| if AdminConfig.getObjectType( id ) == 'Cell' :
84| nodeName = System.getProperty( 'local.node' )
85| else :
86| nodeName = getAttributeValue( id, 'name' )
87| return AdminTask.getNodeBaseProductVersion(
88| '[-nodeName %s]' % nodeName
89| )
90|def WASprofileName( id ) :
91| result = WASvarLookup( id, 'USER_INSTALL_ROOT' )
92| if result :
93| result = result.split( os.sep )[ -1 ]
```

```
94| return result
95|def WAShome( id ) :
96| return WASvarLookup( id, 'WAS_INSTALL_ROOT' )
97|def WASvarLookup( id, name ) :
98| VSE = AdminConfig.list( 'VariableSubstitutionEntry', id )
99| for var in VSE.splitlines () :
100| if getAttributeValue( var, 'symbolicName' ) == name : 101| result =
getAttributeValue( var, 'value' ) 102| break
103| else :
104| result = None
105| return result
```

Table 22-2 provides a description of each of these routines.

Table 22-2. *Part 2 of the WASports_05.py Utility Routines, Explained* **Lines Description**

82–89 The WASversion(...) routine is called when a tree node that corresponds to the cell or a WebSphere node is selected. A string is returned that identifies the product version of the cell or WebSphere node. 90–94 The WASprofileName(...) routine determines and returns the name of the WebSphere profile associated with the specified configuration ID.
95–96 The WAShome(...) routine determines and returns the WebSphere installation directory. 97–105 The WASvarLookup(...) routine determines the value of the specified WebSphere environment variable.
New Classes for WASports_05

Listing 22-14 shows the revised and simplified cellTSL class that includes the actions to be taken when tree selection events occur. The most significant change is related to the last parameter specified on the constructor, which has changed from a dictionary to a cellInfo object.

Listing 22-14. The WASports_05.py Revised cellTSL Class

```
106|class cellTSL( TreeSelectionListener ) :
107| def __init__( self, tree, pane, data ) :
108| self.tree = tree
109| self.pane = pane
110| self.data = data
111| def valueChanged( self, tse ) :
```

```
112| pane = self.pane
113| node = self.tree.getLastSelectedPathComponent()
114| if node :
115| if node.isLeaf() :
116| text = leafFormatString % (
117| node.getParent(),
118| node
119| )
120| else :
121| text = self.data.getInfoValue( node.toString() )
122| else :
123| text = '<html><br/><b>Nothing selected<b/>'
124| pane.setText( text )
```

Listing 22-15 shows the cellInfo class that is instantiated by the cellTree method (not shown) and that's used by the TreeSelectionListener class shown in Listing 22-14.

Listing 22-15. The WASports_05.py Inner cellInfo Class

```
209| class cellInfo :
210| def __init__( self ) :
211| self.names = {} # Dict[ name ] -> configId
212| self.info = {} # Dict[ name ] -> node info
213| def getNames( self ) :
214| return self.names
215| def setNames( self, names ) :
216| self.names = names
217| def addInfoValue( self, index, value ) :
218| self.info[ index ] = '<html>' + (
219| value.replace( '&', '&amp;' ).replace( '<',
220| '&lt;' ).replace( '>', '&gt;' ).replace( ' ',
221| ' ' ).replace( '\n', '<br/>' )
222| )
223| def getInfoValue( self, index ) :
224| return self.info[ index ]
```

Step 6: Displaying Server Port Number Information

The next logical step is to add code to display information about the selected

server in the right side of the split pane. Let's take a quick look at the result of these changes. Figure 22-7 shows the initial attempt of displaying the table of port numbers and their associated named endpoints. Even though it isn't perfect, it does show that you're going in the right direction.

Figure 22-7. *Sample output of the WASports_06.py script*

To add this functionality to this iteration of the script, start by recognizing the possible value of the firstNamedConfigType(...) routine, shown in Listing 22-16. It can be used to simplify calls elsewhere in the script. It will return the first configuration ID matching the specified values or None.

Listing 22-16. New Utility Routine for WASports_06.py

```
53|def firstNamedConfigType(
54| Type, value, scope = None, attr = None
55|) :
56| items = findScopedTypes( Type, value, scope, attr )
57| if len( items ) :
58| result = items[ 0 ]
59| else :
60| result = None
61| return result
```

Listing 22-17 shows the modified TreeSelectionListener class this is now used to display the appropriate information about the selected tree item, including the JTable instance that is returned by the call to the getPortTable(...) method that starts on line 135.

Listing 22-17. Modified cellTSL (TreeSelectionListener) Class

```
122|class cellTSL( TreeSelectionListener ) :
123| def __init__( self, tree, pane, data ) :
124| self.tree = tree
125| self.pane = pane # Reference to splitpane
126| self.data = data
127| def valueChanged( self, tse ) :
128| pane = self.pane
129| loc = pane.getDividerLocation()
130| node = self.tree.getLastSelectedPathComponent()
131| if node :
132| if node.isLeaf() :
133| pane.setRightComponent (
134| JScrollPane(
135| self.data.getPortTable(
136| (
137| str( node.getParent() ),
138| str( node )
139| )
140| )
141| )
142| )
143| else :
144| pane.setRightComponent (
145| JScrollPane(
146| JLabel(
147| self.data.getInfoValue( str( node ) ),
148| font = MONOFONT
149| )
150| )
151| )
152| else :
153| pane.setRightComponent (
154| JScrollPane(
155| JLabel(
156| '<html><br/><b>Nothing selected<b/>', 157| font = MONOFONT
158| )
159| )
```

```
160| )
161| pane.setDividerLocation( loc )
```

A new PortLookupTask class, shown in Listing 22-18, was created to perform the necessary calls to the AdminConfig scripting object in order to create a JTable instance of the port numbers and associated named endpoints (shown on lines 267- 270). When the task finishes, the data object, which is an instance of the cellInfo class, is populated with the appropriate server port table data.

Listing 22-18. Part 1 of the New PortLookupTask Class

```
246|class PortLookupTask( SwingWorker ) :
247| lock = threading.Lock()
248| def __init__(
249| self,
250| nodeName,
251| serverName,
252| data
253| ) :
254| self.nodeName = nodeName
255| self.servName = serverName
256| self.data = data
257| SwingWorker.__init__( self )
258| def doInBackground( self ) :
259| self.lock.acquire()
260| try :
261| pDict = self.getPorts( self.nodeName, self.servName )
262| ports = pDict.keys()
263| ports.sort( lambda x,y: cmp( int( x ), int( y ) ) )
264| result = []
265| for port in ports :
266| result.append( [ port, pDict[ port ] ] )
267| table = JTable(
268| PortTableModel( result ),
269| autoResizeMode = JTable.AUTO_RESIZE_OFF
270| )
271| table.getTableHeader().setReorderingAllowed( 0 )
272| self.data.addPortTable(
273| ( self.nodeName, self.servName ),
```

```
274| table
275| )
276| except :
277| print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
278| self.lock.release()
```

The remainder of this class is shown in Listing 22-19. Remember that the doInBackground(...) method will call the done(...) method when it is complete. In this case, this method doesn’t have to do any additional processing so it only contains the pass statement. The method is left as a placeholder to remind you to consider if additional actions need to be performed.

Listing 22-19. Part 2 of the New PortLookupTask Class

```
279| def done( self ) :
280| pass
281| def getPorts( self, nodeName, serverName ) :
282| scope = firstNamedConfigType( 'Node', nodeName )
283| serverEntry = firstNamedConfigType(
284| 'ServerEntry',
285| serverName,
286| scope,
287| 'serverName'
288| )
289| result = {}
290| if serverEntry :
291| nEPs = AdminConfig.list(
292| 'NamedEndPoint',
293| serverEntry
294| )
295| for namedEndPoint in nEPs.splitlines() :
296| Name = getAttributeValue(
297| namedEndPoint,
298| 'endPointName'
299| )
300| epId = getAttributeValue(
301| namedEndPoint,
302| 'endPoint'
303| )
304| port = getAttributeValue( epId, 'port' )
```

```
305| result[ port ] = Name
306| return result
```

Listing 22-20 shows the PortTableModel class used to hold the data for each server in the cell. This table model class isn’t quite complete, but it is a reasonable start. It identifies the data type for each of the two columns, it identifies only the first column (column 0) as editable, and it includes range checking on the user-supplied port number values.

Listing 22-20. New PortTableModel Class for Holding Port Table Data

```
307|class PortTableModel( DefaultTableModel ) :
308| headings = 'Port#,EndPoint Name'.split( ',' )
309| def __init__( self, data ) :
310| for row in range( len( data ) ) :
311| data[ row ] = [
312| int( data[ row ][ 0 ] ), data[ row ][ 1 ]
313| ]
314| DefaultTableModel.__init__( self, data, self.headings ) 315| def
getColumnClass( self, col ) :
316| if col == 0 :
317| return Integer
318| else :
319| return String
320| def isCellEditable( self, row, col ) :
321| return col == 0
322| def setValueAt( self, value, row, col ) :
323| if 0 <= value <= 65535 :
324| index = ( self, row )
325| DefaultTableModel.setValueAt( self, value, row, col ) 326| else :
327| DefaultTableModel.setValueAt(
328| self,
329| self.getValueAt( row, col ),
330| row,
331| col
332| )
333| self.fireTableCellUpdated( row, col )
```

Step 7: Computing Table Column Widths

Next, you'll see that by adding a small amount of code, you can improve the appearance of the application. Figure 22-8 shows the result of adding a routine to determine the preferred widths of the table columns.

Figure 22-8. *Sample output of the WASports_07.py script*

Listing 22-21 shows the setColumnWidths(...) method that was added to the PortLookupTask class. It is brief because it knows about the table contents. For example, rather than looking for the widest value in column 1, the preferred width for this column is determined using the largest allowed value (65535) for this column.

Listing 22-21. The setColumnWidths(...) Method Added to WASports_07.py

```
311| def setColumnWidths( self, table ) :
312| tcm = table.getColumnModel() # Table Column Model
313| data = table.getModel() # To access table data
314| margin = tcm.getColumnMargin() # gap between columns
315| render = table.getCellRenderer( 0, 0 )
316| comp = render.getTableCellRendererComponent(
317| table, # table being processed
318| '65535', # max port number
319| 0, # not selected
```

```
320| 0, # not in focus
321| 0, # row num
322| 0 # col num
323| )
324| cWidth = comp.getPreferredSize().width
325| col = tcm.getColumn( 0 )
326| col.setPreferredWidth( cWidth + margin )
327| cWidth = -1
328| for row in range( data.getRowCount() ) :
329| render = table.getCellRenderer( row, 1 )
330| comp = render.getTableCellRendererComponent(
331| table,
332| data.getValueAt( row, 1 ), # cell value
333| 0, # not selected
334| 0, # not in focus
335| row, # row num
336| 1 # col num
337| )
338| cWidth = max(
339| cWidth,
340| comp.getPreferredSize().width
341| )
342| col = tcm.getColumn( 1 )
343| col.setPreferredWidth( cWidth + margin )
```

Step 8: Adding Menu Items

What do the changes in this iteration do to the application? Figure 22-9 shows the results of these changes. You can see the menu items that were added.

Figure 22-9. *Sample menu items from the WASports_08.py script*

Listing 22-22 shows the new MenuBar(...) method that was added to the WASports class to create the menu and specify the corresponding event-handling

routines for each menu item. One thing that is important to notice is the fact that the Changes menu entry is initialized as disabled (as you can see from the first image in Figure 22-9). This iteration of the script doesn't include code to enable this menu entry, so you won't be able to see the Save and Discard menu items until a future iteration of the script.

Listing 22-22. New MenuBar(...) Method in the WASports Class from WASports_08.py

```
457| def MenuBar( self ) :
458| menu = JMenuBar()
459| self.ChangesMI = JMenu( 'Changes', enabled = 0 )
460| self.ChangesMI.add(
461| JMenuItem(
462| 'Save',
463| actionPerformed = self.save
464| )
465| )
466| self.ChangesMI.add(
467| JMenuItem(
468| 'Discard',
469| actionPerformed = self.discard
470| )
471| )
472| jmFile = JMenu( 'File' )
473| jmFile.add( self.ChangesMI )
474| jmFile.add(
475| JMenuItem(
476| 'Exit',
477| actionPerformed = self.Exit
478| )
479| )
480| menu.add( jmFile )
481| jmHelp = JMenu( 'Help' )
482| jmHelp.add(
483| JMenuItem(
484| 'About',
485| actionPerformed = self.about 486| )
487| )
```

```
488| jmHelp.add(
489| JMenuItem(
490| 'Notice',
491| actionPerformed = self.notice 492| )
493| )
494| menu.add( jmHelp )
495| return menu
```

Listing 22-23 shows the initial implementations of the event handler methods for the new menu items. One important thing to note is the reference by the about(...) method to the aboutTask.getResult() method. A new AboutTask class was also added to this iteration of the script in order to process the script docstring[6] and create an HTML string that displays nicely. The reason a separate thread task was used is to allow the application to perform the string conversion processing off the main thread, without causing the entire application to pause.

Listing 22-23. The New Event Handlers Added to WASports_08.py

```
496| def about( self, e ) :
497| JOptionPane.showMessageDialog(
498| self.frame,
499| JLabel(
500| self.aboutTask.getResult(),
501| font = MONOFONT
502| ),
503| 'About',
504| JOptionPane.PLAIN_MESSAGE
505| )
506| def notice( self, e ) :
507| JOptionPane.showMessageDialog(
508| self.frame,
509| Disclaimer,
510| 'Notice',
511| JOptionPane.WARNING_MESSAGE
512| )
513| def save( self, e ) :
514| print 'save() - Not yet implemented'
515| def discard( self, e ) :
516| AdminConfig.reset()
```

```
517| def Exit( self, e ) :
518| sys.exit()
```

[6]See https://www.python.org/dev/peps/pep-0257/.

Step 9: Implementing Save and Discard

Now that some menu items are in place, it is reasonable to implement the event handlers and associated code to allow these menu items to be used. Figure 22-10 shows that when a port number value changes, the Changes menu item is enabled, which allows the user to save or discard the changes. Additionally, a dialog box is displayed when the user tries to exit the application without saving or tries to discard their changes.

Figure 22-10. *Sample menu items from the WASports_08.py script*

Listing 22-24 shows the appCleanup(...) routine that displays a confirm dialog box when the user tries to exit the application when unsaved changes exist. If the user doesn't want to discard the changes, control is returned to the caller, which is responsible for resuming operation.

Listing 22-24. The appCleanup() Routine Checks for Unsaved Changes from WASports_09.py

```
79|def appCleanup( app ) :
80| if AdminConfig.hasChanges() :
81| answer = JOptionPane.showConfirmDialog(
82| app, 'Save changes?'
83| )
84| if answer == JOptionPane.YES_OPTION :
85| AdminConfig.save()
86| elif answer in [
```

```
87| JOptionPane.CLOSED_OPTION,
88| JOptionPane.CANCEL_OPTION
89| ] :
90| return
91| else :
92| AdminConfig.reset()
93| print '\nConfiguration changes discarded.'
94| System.gc() # call Java Garbage Collector
95| time.sleep( 0.5 ) # Slight delay for garbage pickup
96| sys.exit( 0 )
```

Listing 22-25 shows the new SaveTask and DiscardTask classes, which are used to perform the potentially long-running AdminConfig.save() or AdminConfig.reset() as a background task. In addition, the DiscardTask is responsible for updating the port number values in the tables on which the changes were made. **Listing 22-25.** The SaveTask and DiscardTask Classes from WASports_09.py

```
311|class SaveTask( SwingWorker ) :
312| def __init__( self, cellData ) :
313| self.cellData = cellData
314| def doInBackground( self ) :
315| try :
316| original = self.cellData.clearOriginals()
317| AdminConfig.save()
318| except :
319| print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
320|class DiscardTask( SwingWorker ) :
321| def __init__( self, cellData ) :
322| self.cellData = cellData
323| def doInBackground( self ) :
324| try :
325| original = self.cellData.getOriginal()
326| tables = []
327| for index in original.keys() :
328| table, row = index
329| table.getModel().resetPortValue(
330| row,
331| original[ index ]
```

```
332| )
333| if table not in tables :
334| tables.append( table )
335| for table in tables :
336| table.repaint()
337| AdminConfig.reset()
338| except :
339| print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
```

Listing 22-26 required changes in order to perform the actual AdminConfig.modify(...) method (see line 470), which uses the AdminConfig scripting object to change the specified port number for the indicated named endpoint on the user-selected application server. It is also responsible for calling the cellInfo addOriginal(...) method that’s used to save the original value should the Discard menu entry be called.

Listing 22-26. Part 1 of the Modified PortTableModel Class from WASports_09.py

```
444|class PortTableModel( DefaultTableModel ) :
445| headings = 'Port#,EndPoint Name'.split( ',' )
446| def __init__( self, data ) :
447| self.table = None
448| self.nodeName = None
449| self.serverName = None
450| self.epIdDict = None
451| self.app = None
452| for row in range( len( data ) ) :
453| data[ row ][ 0 ] = int( data[ row ][ 0 ] )
454| DefaultTableModel.__init__( self, data, self.headings ) 455| def
getColumnClass( self, col ) :
456| if col == 0 :
457| return Integer
458| else :
459| return String
460| def isCellEditable( self, row, col ) :
461| return col == 0
462| def resetPortValue( self, row, value ) :
463| DefaultTableModel.setValueAt( self, value, row, 0 ) 464|
```

self.fireTableCellUpdated(row, 0)

Listing 22-27 shows the remainder of the PortTableModel class. You are encouraged to search in the script file to see which of these methods is referenced by the rest of the script.

Listing 22-27. Part 2 of the Modified PortTableModel Class from WASports_09.py

```
465| def setValueAt( self, value, row, col ) :
466| prev = self.getValueAt( row, col )
467| if 0 <= value <= 65535 :
468| name = self.getValueAt( row, 1 )
469| epId = self.epIdDict[ name ]
470| AdminConfig.modify( epId, [ [ 'port', value ] ] )
471| self.app.ChangesMI.setEnabled( 1 )
472| DefaultTableModel.setValueAt( self, value, row, col )
473| self.app.cellData.addOriginal( self.table, row, prev )
474| else :
475| DefaultTableModel.setValueAt(
476| self,
477| prev,
478| row,
479| col
480| )
481| self.fireTableCellUpdated( row, col )
482| def getContext( self ) :
483| return self.table, self.nodeName, self.serverName
484| def setContext(
485| self, table, nodeName, serverName, epIdDict, app
486| ) :
487| self.table = table
488| self.nodeName = nodeName
489| self.serverName = serverName
490| self.epIdDict = epIdDict
491| self.app = app
```

Listing 22-28 shows the changes that need to be made to the cellInfo class to allow the original port number values to be saved when the user modifies a value.

Listing 22-28. The Modified (Inner) cellInfo Class

```
492|class WASports_09( java.lang.Runnable ) :
493| class cellInfo :
| ...
500| def addOriginal( self, table, row, value ) :
501| self.lock.acquire()
502| index = ( table, row )
503| if not self.before.has_key( index ) :
504| self.before[ index ] = value
505| self.lock.release()
506| def getOriginal( self ) :
507| self.lock.acquire()
508| result = self.before
509| self.lock.release()
510| return result
511| def clearOriginals( self ) :
512| self.lock.acquire()
513| self.before = {}
514| self.lock.release()
```

Listing 22-29 defines a WindowAdapter descendent for the application. The event handler method in this class is invoked when the user clicks on the application close icon (the [X] in the upper-right corner of the application).
Listing 22-29. The WindowAdapter Class Handles the windowClosed Events from WASports_09.py

```
650|class windowAdapter( WindowAdapter ) :
651| def windowClosed( self, e ) :
652| frame = e.getWindow()
653| appCleanup( frame )
654| frame.setVisible( 1 ) # User chose cancel or close
```

Step 10: Implementing the Export Functionality

Now you get to some new and interesting[7] stuff that is unrelated to Swing, but directly related to non-trivial applications that need to be developed. Take a few moments to consider how you might implement an Export capability. It is important to realize that one of the most important questions you should ask is,

“What data format should be used?” A number of potential alternatives may come to mind, some being more useful than others.

When I first started thinking about this topic, I considered using XML files to hold the information. This also makes a lot of sense when dealing with WebSphere Application Server configuration files because so many of them use the XML format.

My initial investigations into using the simplest XML module (xml.dom.minidom) didn’t fare well. I tried using a trivial example to test the validity of this as far as the needs of this program were concerned. Listing 22-30 shows a simple interactive wsadmin session and the inability of the xml.dom.minidom module to perform the simplest of XML parsing.

[7]Remember that old curse, “may you live in interesting times?” See http://en.wikipedia.org/wiki/May_you_live_in_ interesting_times.

Listing 22-30. Testing the xml.dom.minidom

```
wsadmin>from xml.dom.minidom import parse
wsadmin>
wsadmin>try :
wsadmin> dom = parse( '70DM.xml' )
wsadmin>except :
wsadmin> print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ] wsadmin>
Failed to get environment, environ will be empty: ...

Error: exceptions.AttributeError value: feed
wsadmin>
```

Does this mean that you can’t use XML to represent the data that the application needs to save (export) or load (import)? Not at all. The xml.dom.minidom module is part of the optional libraries that are provided by wsadmin scripts.[8] Just because they are present doesn’t mean that you have to use them.

What other XML modules and libraries exist? It is important, especially at times like these, to remember that other libraries exist as part of Java J2EE.[9] There are a number of free online resources that explain how to work with XML in Java.[10] I spent some time reading the J2EE tutorial,[11] specifically Chapter 2, “Understanding XML.” There is also a number of very good publications that

discuss XML processing with Java. Unfortunately, the Jython version of those would be a whole separate book, so I'll provide some examples that correspond very closely to the kind of Java examples available elsewhere.

Using the Document Object Model API

The kinds of XML processes discussed in this section use the Document Object Model (DOM) Application Programming Interface (API) and the Simple API for XML (SAX). Of these two, the DOM is much simpler and easier to write and therefore to read and understand. Let's start with it.

Listing 22-31 shows the important part of a simple script that demonstrates how to read an XML file and then process the data structure that is created when the input file is successfully parsed. It is so simple to read and parse an XML file that it can be performed in a single statement (see lines 22-25). The traverse routine on lines 6-19 uses recursion to traverse the data structure that's produced.

Listing 22-31. Simple DOM Routines from xmlDOM.py

```
6|def traverse( node, indent = 0 ) :
7| prefix = '%*s' % ( indent, '' )
8| while node :
9| if node.getNodeType() == Node.ELEMENT_NODE : 10| print '%s<%s>' % (
prefix, node.getNodeName() ) 11| traverse( node.getFirstChild(), indent + 2 ) 12|
print '%s</%s>' % ( prefix, node.getNodeName() ) 13| elif node.getNodeType()
== Node.TEXT_NODE : 14| value = node.getNodeValue()
15| if value :
16| value = value.strip()
17| if value :
18| print '%s"%s"' % ( prefix, value ) 19| node = node.getNextSibling()
20|def dom( filename ):
21| try :
22| doc = DocumentBuilderFactory.newInstance( 23|
).newDocumentBuilder().parse(
24| File( filename )
25| )
26| root = doc.getDocumentElement()
27| root.normalize()
28| traverse( root )
```

```
29| except :
30| print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
```

[8] For example, %WAS_HOME%\optionalLibraries\jython\Lib\xml\dom\minidom.py.
[9]*I admit that I tend to use the J2EE Reference because WebSphere is an Application Server.*
[10]See http://docs.oracle.com/javaee/1.4/tutorial/information/faq.html.
[11]See http://docs.oracle.com/javaee/1.4/tutorial/doc/index.html or http://docs.oracle.com/javaee/1.4/ tutorial/doc/J2EETutorial.pdf.

This example demonstrates one of the greatest drawbacks of the DOM, and that is the fact that parsing an XML file using the DOM API requires the entire contents of the file to be loaded into memory. For the current application, this might not be too much of a restriction if you are working with the information pertaining to a single WebSphere Application Server cell. However, some cells can be quite large, so this might be a limiting factor. Additionally, I can imagine the application being expanded to work with multiple cells, which could significantly increase the memory requirements of the application.

Using the Simple API for XML (SAX)

Just how simple is the Simple API for XML? It's really not too bad at all. It is possible to have a single statement that parses an XML file using the SAX API, but I'm not going to be that mean to myself or others.[12] Listing 22-32 shows a trivial routine from the xmlSAX.py script file that shows how to instantiate a SAX parser and use it to parse a userspecified XML input file.

■ **Note** one of the import differences between these two examples is that this script validates the XMl during the processing. So, in addition to the XMl input, a Document type Definition (DtD) file defines the tags that can exist in a valid XMl file, how they should be used, and their relationship to one another. Unfortunately, details describing and explaining DtDs are beyond the scope of this book.

[12]Just because it is possible to write a one-line statement to perform this task does *not* mean that you should. **Listing 22-32.** The readFileSAX Routine from xmlSAX.py

```
61|def readFileSAX( filename ) :
62| try :
63| FIS = FileInputStream( File( filename ) )
64| ISR = InputStreamReader( FIS, ENCODING )
65| src = InputSource( ISR, encoding = ENCODING )
66| factory = SAXParserFactory.newInstance()
67| factory.setValidating( 1 )
68| parser = factory.newSAXParser()
69| parser.parse( src, SAXhandler() )
70| except :
71| print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
```

Listings 22-33 and 22-34 show the SAXhandler class from the xmlSAX.py file. It is important to note that the SAXhandler methods are called by the SAX parser while the input file is being processed. This allows the script using this class to transform the input data into whatever data structure the developer chooses. Using the DOM technique results in a completely processed XML file being represented in memory using the Document Object Model. If this data structure isn't appropriate for your application needs, you need to transform the DOM to a more appropriate data structure for your specific needs.

Listing 22-33. Part 1 of the SAXhandler class from xmlSAX.py

```
10|class SAXhandler( DefaultHandler ) :
11| def __init__( self ) :
12| self.chars = ''
13| self.prefix = ''
14| self.width = 0
15| def indent( self ) :
16| self.width += 2
17| self.prefix = '%*s' % ( self.width, '' )
18| def dedent( self ) :
19| self.width -= 2
20| self.prefix = '%*s' % ( self.width, '' )
21| def startElement( self, uri, localName, name, attributes ) :
22| if self.chars :
23| print '%s"%s"' % ( self.prefix, self.chars )
24| self.chars = ''
25| attr = [
```

```
26| (
27| attributes.getQName( i ),
28| attributes.getValue( i )
29| )
30| for i in range( attributes.getLength() )
31| ]
32| if attr :
33| print '%s<%s %s>' % (
34| self.prefix,
35| name,
36| ', '.join(
37| [
38| '%s="%s"' % ( n, v ) for n, v in attr 39| ]
40| )
41| )
42| else :
43| print '%s<%s>' % ( self.prefix, name ) 44| self.indent()
```

Listing 22-34 shows the remainder of the SAXhandler class.

Listing 22-34. Part 2 of the SAXhandler Class from xmlSAX.py

```
45| def endElement( self, uri, localName, name ) :
46| if self.chars :
47| print '%s"%s"' % ( self.prefix, self.chars )
48| self.chars = ''
49| self.dedent()
50| print '%s</%s>' % ( self.prefix, name )
51| def characters( self, ch, start, length ) :
52| value = str( String( ch, start, length ) ).strip()
53| if value :
54| self.chars += value
55| def warning( self, e ) :
56| print 'Warning:', e.getMessage()
57| def error( self, e ) :
58| print 'Error:', e.getMessage()
59| def fatalError( self, e ) :
60| print 'Fatal error:', e.getMessage()
```

The SAX parser calls only some of the methods in the SAXhandler class

directly. Specifically, the startElement(...), endElement(...), and characters(...) methods are called during normal parsing of an XML document. The warning(...), error(...), and fatalError(...) methods are called when unexpected input is encountered.

It might not be obvious why the chars attribute is used as a sort of holding buffer. Why not just display the data when the characters(...) method is called? The answer is that there is no guarantee that all of the data between a start and end tag will be processed at one time. Character sequences can be processed in multiple segments. This class buffers the character data until the next start or end tag is encountered. That is why the startElement(...) and endElement(...) methods begin by looking for buffered character data.

The Document Type Definition (DTD)

One of the useful features of the XML parsers is the support provided to validate the input being processed against the description of valid input. That is the role of the Document Type Definition (DTD). It identifies the valid tags and their order, and identifies which tags are allowed to have attributes. Given an appropriate DTD, the parser factory can have validation enabled by calling the setValidating(...) method, as shown in Listing 22-32, line 67. Then, the XML file should identify how the validation should be checked. One way to do this is to include a DOCTYPE definition that identifies whether the DTD is contained in the XML file or in an external file, as shown in Figure 22-11.

```
<!DOCTYPE WASports SYSTEM "WASports.dtd">
```

Figure 22-11. *DOCTYPE definition identifying an external DTD file*

Initial WASports DTD

Using the information contained in the cellInfo class, you can describe the format of an XML file using the DTD shown in Listing 22-35.[13] If you aren't familiar with the syntax of the DTD file, information is available in a number of places on the Internet[14] and in publications about XML.

Listing 22-35. Initial WASports.dtd

```
1|<!ELEMENT WASports (cell)>
2|<!ATTLIST WASports version CDATA #REQUIRED>
3|<!ELEMENT cell (name,WAShome,WASversion,profile,node+)>
```

```
4|<!ELEMENT node (name,WAShome,WASversion,profile,
|hostname+,ipaddr,server+)>
5|<!ELEMENT server (name,endpoint+)>
6|<!ELEMENT name (#PCDATA)>
7|<!ELEMENT WAShome (#PCDATA)>
8|<!ELEMENT WASversion (#PCDATA)>
9|<!ELEMENT profile (#PCDATA)>
10|<!ELEMENT hostname (#PCDATA)>
11|<!ELEMENT ipaddr (#PCDATA)>
12|<!ELEMENT endpoint (name,port)>
13|<!ELEMENT port (#PCDATA)>
```

Table 22-3 describes the DTD in Figure 22-11, line by line. Using the description and the WASports_10.py application, you should be able to better understand the correlation between the application output and the DTD format.

[13]Note that line 4 is too long to fit in the available space and is continued on the next line. [14]For example, see http://www.w3schools.com/dtd/dtd_intro.asp.

Table 22-3. WASports.dtd, *Explained*

Lines	**Description**
1	The root tag is WASports and it has only one cell tag.
2	The WASports tag has a required attribute called version.
3	The cell tag requires a name, WAShome, WASversion, profile, and one or more node tags.
4	The node tag requires a name, WAShome, WASversion, profile, one or more hostnames, one ip_addr and one or more server tags.
5	Each server tag requires one name and one or more endpoint tags.
12	Each endpoint tag requires a name and port tag.
6–13	The tags identified as having (#PCDATA) require some character data.

The ExportTask Class

You need to make some changes to the application menu structure, in order to provide the user with a way to specify that an export should occur. Creating the menu is simple enough that it doesn’t need to be shown here. However, the event handler that is called requires a bit more explanation, so the Export(...) method is

shown in Listing 22-36.

Listing 22-36. Export(...) Method from WASports_10.py

```
788| def Export( self, event ) :
789| title = 'Export (Save) cell details'
790| fc = JFileChooser(
791| currentDirectory = File( '.' ),
792| dialogTitle = title,
793| fileFilter = XMLfiles()
794| )
795| if fc.showOpenDialog(
796| self.frame
797| ) == JFileChooser.APPROVE_OPTION :
798| f = fc.getSelectedFile()
799| fileName = fc.getSelectedFile().getAbsolutePath()
800| if not fileName.endswith( '.xml' ) :
801| fileName += '.xml'
802| msg = 'Overwrite existing file (%s)?'
803| if os.path.isfile( fileName ) :
804| response = JOptionPane.showConfirmDialog(
805| None,
806| msg % os.path.basename( fileName ),
807| 'Confirm Overwrite',
808| JOptionPane.OK_CANCEL_OPTION,
809| JOptionPane.QUESTION_MESSAGE
810| )
811| if response == JOptionPane.CANCEL_OPTION :
812| return
813| ExportTask( fileName, self.cellData ).execute()
```

This method uses the JFileChooser and FileFilter classes discussed in detail in Chapter 17. It limits the kinds of files displayed by the JFileChooser to those having an .xml extension. If the user selects an existing file, it also verifies that the user wants to replace the existing file using a simple confirmation dialog window. The most important part of this method is when it instantiates and begins execution of an ExportTask thread to actually create the specified file (line 813). Listing 22-37 shows the first of three parts of the ExportTask class.

Listing 22-37. Part 1 of the ExportTask Class from WASports_10.py

```
356|class ExportTask( SwingWorker ) :
357| def __init__( self, fileName, cellData ) :
358| self.fileName = fileName
359| self.cellData = cellData
360| def doInBackground( self ) :
361| try :
362| fos = FileOutputStream( self.fileName )
363| streamResult = StreamResult(
364| OutputStreamWriter( fos, 'ISO-8859-1' )
365| )
366| trans = SAXTransformerFactory.newInstance()
367| trans.setAttribute( 'indent-number', 4 )
368| tHand = trans.newTransformerHandler()
369| serializer = tHand.getTransformer()
370| serializer.setOutputProperty(
371| OutputKeys.ENCODING, 'ISO-8859-1'
372| )
373| serializer.setOutputProperty(
374| OutputKeys.DOCTYPE_SYSTEM, 'WASports.dtd'
375| )
376| serializer.setOutputProperty(
377| OutputKeys.INDENT, 'yes'
378| )
379| tHand.setResult( streamResult )
380| tHand.startDocument()
381| atts = AttributesImpl()
382| atts.addAttribute(
383| '', '', 'version', 'CDATA', __version__
384| )
385| tHand.startElement( '', '', 'WASports', atts )
386| atts.clear()
387| tHand.startElement( '', '', 'cell', atts )
388| data = self.cellData
389| root = data.tree.getModel().getRoot()
390| cellName = root.toString()
391| self.addTagAndText( tHand, 'name', cellName )
392| info = data.getInfoDict( cellName )
393| self.addTagAndText(
394| tHand, 'WAShome', info[ 'WAShome' ]
```

```
395| )
396| self.addTagAndText(
397| tHand, 'WASversion', info[ 'WASversion' ]
398| )
399| self.addTagAndText(
400| tHand, 'profile', info[ 'profile' ] 401| )
```

Listing 22-38 shows the next part of the ExportTask class. Again, you can see that a non-trivial class can’t easily be forced to fit onto one or even two pages of this book.

Listing 22-38. Part 2 of the ExportTask Class from WASports_10.py

```
402| nodes = root.children()
403| while nodes.hasMoreElements() :
404| node = nodes.nextElement()
405| nodeName = node.toString()
406| info = data.getInfoDict( nodeName )
407| tHand.startElement( '', '', 'node', atts )
408| self.addTagAndText( tHand, 'name', nodeName )
409| self.addTagAndText(
410| tHand, 'WAShome', info[ 'WAShome' ]
411| )
412| self.addTagAndText(
413| tHand, 'WASversion', info[ 'WASversion' ]
414| )
415| self.addTagAndText(
416| tHand, 'profile', info[ 'profile' ]
417| )
418| self.addTagAndText(
419| tHand, 'hostname', info[ 'hostnames' ]
420| )
421| self.addTagAndText(
422| tHand, 'ip_addr', info[ 'ipaddr' ]
423| )
424| servers = node.children()
425| while servers.hasMoreElements() :
426| server = servers.nextElement()
427| serverName = server.toString()
428| tHand.startElement(
```

```
429| ", ", 'server', atts
430| )
431| self.addTagAndText(
432| tHand, 'name', serverName
433| )
434| table = data.getPortTable(
435| ( nodeName, serverName )
436| )
437| model = table.getModel()
438| for row in range( model.getRowCount() ) :
439| tHand.startElement(
440| ", ", 'endpoint', atts
441| )
442| name = model.getValueAt( row, 1 )
```

Listing 22-39 shows the remainder of the ExportTask class. Unfortunately, breaking a listing like this across multiple pages can make it difficult to read. Therefore, you are encouraged to use your favorite text editor and view the script source file.

Listing 22-39. Part 3 of the ExportTask Class from WASports_10.py

```
443| self.addTagAndText(
444| tHand, 'name', name
445| )
446| port = str(
447| model.getValueAt( row, 0 )
448| )
449| self.addTagAndText(
450| tHand, 'port', port
451| )
452| tHand.endElement(
453| ", ", 'endpoint'
454| )
455| tHand.endElement( ", ", 'server' )
456| tHand.endElement( ", ", 'node' )
457| tHand.endElement( ", ", 'cell' )
458| tHand.endElement( ", ", 'WASports' )
459| tHand.endDocument()
460| except :
461| msgText = '\nExportTask() Error: %s\nvalue: %s'
```

```
462| print msgText % sys.exc_info()[ :2 ]
463| def addTagAndText( self, handler, tagName, text ) :
464| handler.startElement(
465| ", ", tagName, AttributesImpl()
466| )
467| handler.characters(
468| String( text ).toCharArray(), 0, len( text )
469| )
470| handler.endElement( ", ", tagName )
```

The ExportTask shown in Listings 22-37 through 22-39 is described in detail in Table 22-4. The important point to note about the ExportTask class is that every XML tag calls the startElement(...) method, the required nested elements, and the endElement(...) method in order to create a properly formed XML document.

Table 22-4. *ExportTask Class, Explained*

Lines	Description
357–359	Class constructor used to save the specified filename and cellInfo structure containing the information to be exported.
362–365	The output file stream is instantiated.
366–367	A SAXTransformerFactory instance is created with the specified indentation (for readability).
368–379	A transformer handler instance is created with the specified properties (such as encoding). It is also associated with the output stream created earlier.
380	The startDocument(...) method begins to create the XML document being produced.
381–385	The WASports tag is created with the required version attribute.
386	Since no other tags in the document have attributes, the atts variable is cleared so it can be reused on other tags.
387	The only instance of the cell tag is started.
388–401	The tags required to be present in the cell tag hierarchy are created using the addTagAndText(...) utility method.
402–456	Each node in the cell has its required tags and all of the associated server entries are created.

457–459 Note how each startElement has a matching endElement after all the contained tags have been created.
460–462 An exception clause is invoked when an error is encountered. Unfortunately, it is a trivial error handler.

463–470 The addTagAndText(...) utility method simplifies the creation of tags like <tagName>text value</tagName>.

Step 11: Implementing the Import Functionality

One of the most challenging aspects of the iterative development of non-trivial applications such as WASports is that you can reach a point where significant changes are required for the next iteration. Step 11 is where these changes are recognized. How do I know that? Let's take some time to consider the implications associated with being able to import application information. To make it more interesting, you'll do so by using a few question and answers to get things started.

Table 22-5. *Important Questions and Answers*

Q: What should occur when a WASports file is imported?
A: A new internal frame should be created similar in nature to the one that is created when the application starts. Q: How will an imported frame differ from an existing one?
A: These frames can be closed, and if changes are made, a save needs to be performed using the export functionality, instead of changing the current configuration.
Q: Do changes to existing classes and data structures need to be made?
A: Yes; for example, the internal cellInfo class needs to be made global so that both local cell data and imported cell data can use the same class. Additionally, the setColumnWidths(...) method should be moved from within the PortLookupTask class to the global scope so that it can also be called by the ImportTask class. Additionally, the names dictionary in the cellInfo class can't refer to an actual configuration ID because you don't want any of the changes made to imported data to be made in the local configuration. So, for the imported data, a pseudoconfigId will be created instead.
Q: What kinds of things needed to be "fixed" because of improved understanding and how should they be handled? A: During this iteration, it became clear that the internal form of the JOption dialog boxes should have been

used instead of the default external form (that is, the JOptionPane.showInternalConfirmDialog(...) method instead of the JOptionPane.showConfirmDialog(...) method). Additionally, using an ip_addr tag for the ipaddr entry in the cellInfo class added unnecessary complexity. So the DTD and code was changed to use an ipaddr tag instead.

The ImportTask Class

The changes required to add the Import functionality are similar to the changes needed to add the Export functionality discussed previously. The Import(...) event handler method[15] isn't shown here, but can be found in the WASports_11.py script file.[16] It too uses a JFileChooser instance to allow the user to identify the XML file to be imported. Once the input file is identified, an ImportTask instance is created to perform the input processing on a separate thread. Listing 22-40 shows the ImportTask class used by this iteration of the WASports script.

Listing 22-40. Part 1 of the ImportTask Class from WASports_11.py

```
780|class ImportTask( SwingWorker ) :
| ...
992| def __init__( self, app, menuItem, fileName ) :
993| self.app = app
994| self.menuItem = menuItem
995| self.fileName = fileName
996| self.handler = ImportTask.SAXhandler( app )
997| menuItem.setEnabled( 0 )
998| self.msgText = ''
999| def doInBackground( self ) :
1000| try :
1001| FIS = FileInputStream( File( self.fileName ) ) 1002| ISR =
InputStreamReader( FIS, ENCODING ) 1003| src = InputSource( ISR, encoding
= ENCODING ) 1004| factory = SAXParserFactory.newInstance() 1005|
factory.setValidating( 1 )
1006| parser = factory.newSAXParser()
1007| parser.parse( src, self.handler )
1008| FIS.close()
1009| except :
1010| msgText = 'Error: %s\nvalue: %s'
```

```
1011| self.msgText = msgText % sys.exc_info()[ :2 ] 1012| def done( self ) :
1013| localFrame = self.app.localFrame
1014| desktop = self.app.frame.getContentPane() 1015| errors =
self.handler.errors
1016| if self.msgText or errors :
1017| if self.msgText :
1018| msg = self.msgText + '\n'
1019| else :
1020| msg = ''
1021| for error in self.handler.errors :
1022| msg += ( '\n' + error )
1023| if msg.startswith( '\n' ) :
1024| msg = msg[ 1: ]
1025| JOptionPane.showInternalMessageDialog( 1026| desktop,
1027| msg,
1028| 'Import failed',
1029| JOptionPane.ERROR_MESSAGE,
1030| None
1031| )
```

[15]Note the use of capitalization to differentiate this method name from the import keyword. [16]The complete source can be found in ...\code\Chap_22\WASports_11.py.

Listing 22-41 continues the ImportTask class started in the previous listing.

Listing 22-41. Part 2 of the ImportTask Class from WASports_11.py

```
1032| else :
1033| cellData = self.handler.getResults()
1034| tree = cellData.getTree()
1035| root = tree.getModel().getRoot()
1036| name = root.toString()
1037| if localFrame :
1038| size = localFrame.getSize()
1039| w, h = size.width, size.height
1040| count = len( self.app.frames )
1041| num = ( count - 1 ) % 8
1042| loc = Point(
1043| num * 27 + 32,
1044| num * 27 + 32
```

```
1045| )
1046| else :
1047| print '\nWarning: no localFrame' 1048| size = self.app.frame.getSize()
1049| w, h = size.width >> 1, size.height >> 1 1050| count = 0
1051| loc = Point( 32, 32 )
1052| internal = InternalFrame(
1053| title = '%d : %s' % ( count, name ), 1054| size = Dimension( w , h ),
1055| location = loc,
1056| cellData = cellData,
1057| app = self.app,
1058| closable = 1
1059| )
1060| self.app.frames.append(
1061| ( internal, self.fileName )
1062| )
1063| desktop.add( internal, None, 0 )
1064| internal.setSelected( 1 )
1065| node = tree.getModel().getRoot()
1066| tree.expandPath( TreePath( node.getPath() ) ) 1067|
self.menuItem.setEnabled( 1 )
```

It is interesting to see how much of this code deals with potential problems. This is one of the temptations that exist with rapid prototyping and the iterative development of applications. There is a tendency to leave out error checking and diagnostic or informational messages.

Unfortunately, this choice has its consequences, which are, by definition, self-inflicted and often painful. When you don't include error checking and diagnostic messages as you're developing your application, it's much more difficult to determine the source of any problems that crop up.[17]

Most of the code in Listings 22-40 and 22-41 should look familiar. For example, compare the doInBackground(...) method (lines 1001-1008) with the code in Listing 22-27. Both prepare the input stream using the appropriate encoding and instantiate a validating SAXParserFactory instance. They also both use this parser to process the specified input stream using a SAXhandler instance that has been customized for the application.

In this case, the SAXhandler class isn't shown, but is very similar, at least in

structure, to the one in Listing 22-33. The biggest difference is that instead of displaying the buffered character strings, they are saved in a cellInfo dictionary indexed by the associated tag name (the cell name is in an entry indexed by cellName).

The code is simplified by the fact that the parser validates the input against the DTD and calls one of the errorrelated routines if an error condition is detected (the warning(...), error(...), or fatalError(...) methods).
At the appropriate points in the parsing process (when the node startElement event is detected), all of the details required by the preceding hierarchy level (in this case, between the cell tag and the node tag) have been saved and can now be used to add a new tree element.
When the end of the XML document is encountered and no error conditions have been detected, the done(...) method process shown in Listing 22-40 uses the result of the processing to create a new inner frame instance and display it on the application desktop.

[17]Personal note: I learned the hard way that SwingWorker threads can fail silently (exceptions aren't displayed), so it is good to consider using try/except blocks in these kinds of threads.

What's Left?

At this point, the script is a realistic demonstration of the kinds of things that you can do to create a reasonable graphical interactive wsadmin application. Of course, there are features that you could add to make the application even more useful. The next sections discuss what you would need to do to add functionality to the script.

Text Highlighting

What would it take to add text highlighting to the application? It depends on what you want to highlight. For example, if you want to highlight the port table rows that contain specific text or specific port number values, you need to use a TableCellRenderer (see Chapter 12) to control how each cell should be rendered. Be careful, though; it is easy to forget that the isSelected argument of the getTableCellRendererComponent(...) method should be involved with determining how the table cell should be rendered.

Another possibility is to allow the user to highlight those table rows for active ports. However, remember to take into account the hostname or IP address of the local machine when determining whether imported port table rows should be highlighted.

There is also the possibility of adding multiple or combined highlighting conditions at the same time (both active and highlighted because of matching endpoint names or port numbers).
Maybe you want to allow the user to dynamically select the colors to be used for highlighting. If so, how would you allow the user to save this kind of preference information?

Table Sorting

It would be somewhat helpful to be able to sort and reorder the port tables. It isn't too difficult to implement,[18] but the potential value is more difficult to judge. This is true because it is hard to guess how valuable being able to reorder the data will be to the users of your application.

Comparing Configurations

One interesting possibility to consider is related to the prospect of using this kind of application to compare two (or possibly more) configurations. What kinds of comparisons would you like to make? You could use an application like this to identify port conflicts and differences.

Wouldn't it be useful if the application could export the current port details and then import the previous configuration details and highlight the differences? Would you want it to be able to copy the port numbers from an imported XML file and use this information to change the values used in the local configuration? This kind of thing might be very useful if you wanted the environments to use identical port numbers.

Report Generation

Another potentially useful feature would be the capability to generate reports about the port numbers being used in one or more WebSphere environments. How might you want these reports organized? How important would it be for the data in these reports to be aggregates or filtering of the available data?

[18]See http://docs.oracle.com/javase/tutorial/uiswing/components/table.html#sorting.

Summary

What next step makes the most sense to you? It depends on your specific needs. Can you use this application as it is? Maybe, maybe not; it depends on your needs. Consider using an application like this to view and manage all of the port numbers being used by your environment. Which would you prefer—to use an application like this or to use the WebSphere administration console to identify and manage the port numbers in your environment?

I think that you'll agree that this kind of application has real potential for making the administration of your WebSphere Application Server environment much simpler.
I hope that this book helps you on your journey of creating useful graphical wsadmin scripting applications.

Index

about(…) method, 443
aboutTask.getResult() method, 443
Absolute Layout Manager

advantages, 36
application frame, 36
frame.add(), 36
getPreferredSize() method, 36
repaint() method, 35
setBounds() method, 36

Absolute layout technique, 222
AbstractAction class, 203
AbstractCellEditor class, 178
Accelerator, 132
ActionEvent, 205
ActionListener class, 126

ActionListener event handler, 31, 224
ActionListener method, 214
actionPerformed() method, 30–31, 47
Adapter classes, 209
addActionListener() method, 32
addButtons() method, 42–43
addCards() method, 43
addComponents() method, 61
add() method, 18
addTab() method, 51
AdminConfig methods, 423
AdminConfig.modify(…) method, 445
Administration console, 385, 419
AdminTask commands, 387
AdminTask.generateSecConfigReport() method, 416 AdminTask.help(…) method, 379
AdminTask.listServerPorts() method, 420
AdminTask methods, 420
AdminTask.reportConfiguredPorts() method, 422 appCleanup(…) routine, 444
Application Programming Interface (API)
DOM, 448
XML (SAX), 449

AutoResizeMode, 188

Bindings
AbstractAction class, 203
boundary conditions, 203
definition, 193
inputMaps and actionMaps

getters and setters, 195
JComponent map attributes, 194 JTable hierarchy, 196
JTable map methods, 194

Boolean cell editors, 172
BorderLayout Manager
advantages, 39

application, 38

BorderLayoutGap.py script, 40
BorderLayoutNEWS.py script, 40
component separation, 40
directional constants, 38–39
disadvantages, 39
horizontal and vertical gaps identification, 41

Box class strut methods, 56
Box.createHorizontalBox() method, 54
BoxLayoutDemo sample application, 51 BoxLayout Manager

Box class, 53
boxes and resizable components, 57 BoxLayoutDemo class, 52–53
BoxLayoutDemo sample application, 51 constants, 53
constructor, 54
createHorizontalBox() method, 54
createVerticalBox() method, 54
invisible box components

BoxLayout Manager (*cont.*)
glue, 55
rigid area, 57
strut, 56

tabs, 51–52
Button
ActionListener class, 30–31
ActionListener event handler, 31
actionPerformed() method, 31
addActionListener() method, 32
frame.add() method, 33
java.awt.event.ActionListener classes, 30
java.lang.Runnable class, 30
multiple inheritance, 30
trivial Java application, 29
button() method, 47
buttonPress() event-handling method, 43

CardLayout Manager
addButtons() method, 42
addCards() method, 43
application, 41–42
buttonPress() event-handling method, 43 buttonPress() method, 42–43
event.getActionCommand() method, 43 JPanel instances, 42
Layout Manager's show() method, 43
run() method, 42

Cell editors
Boolean cell editors, 172
custom numeric cell editors, 174
JComboBox cell editors, 175
numeric cell editors, 173

cellInfo addOriginal(…) method, 445

Cell renderers
column-specific cell renderer, 166
custom cell renderers, 163
custom renderer

Boolean values, 165
TableSelection output, 164
data type rendering, 163
data type-specific cell renderers, 162
DefaultTableCellRenderer class, 165, 169 getTableCellRendererComponent(…) method, 165 setCellRenderer(…) method, 166
TableCellRenderer interface, 165
TableColumnModel, 166
cellTree() method, 427, 430
cellTSL class, 428
ChangeListener method, 363
characters(…) method, 451
class run() method, 13
Column manipulation, 183
column adjustments, 188

column widths, 183
auto resize mode, 189
column heading width, 184 determination, 185
getValueAt(…) method, 186 setColumnWidths(…) method, 186
ComponentEventDemo.java, 217
componentResized() method, 221
Components, 1
consoleTimeout script
GUI Jython version
AdminConfig methods, 339 output, 339
run() method, 337
SwingWorker class, 339–340 update () method, 338
non-GUI Jython version, 336
Container add() method, 23
createEmptyBorder(…) method, 300 Cursor class
isSelected() method, 298
WaitCursor1.py script, 297–298 CustomDialog class, 273
Custom numeric cell editors, 174

DefaultCellEditor, 174–175, 179–180
DefaultTableCellRenderer, 162
DefaultTableModel, 162
Dialog boxes

CustomDialog class, 273
definition, 263
GraphicsConfiguration

component, 265
JOptionPane methods, 274
multiple modal dialog boxes, 273
SimpleDialog class, 270

Document Object Model (DOM), 448
Document Type Definition (DTD), 452 doInBackground(…) method, 302, 305, 439, 460 done(…) method, 439

E

endElement(…) method, 451, 456
event.getActionCommand() method, 43, 59 Event handler method, 443
expandtabs() method, 355
Export(…) method, 453

F

FileFilter class, 454
findScopedTypes(…) routine, 434 firstNamedConfigType(…) routine, 436
FlowLayout Manager, 37
FormattingVisitor() method, 248 frame.add() method, 23
frame.getBounds() method, 221 Frame resize method, 400

G

generateSecConfigReport, 388
getAttributeValue(…) routine, 434
getContentPane() method, 18, 300
getFontMetrics(…) method, 371
getHeaders class, 246
getHostnames(…) routine, 434
getIPaddresses(…) routine, 434
getMonths() method, 93
getPortTable(…) method, 437
getPreferredSize() method, 36
getTableCellEditorComponent(…) method, 171
getTableCellRendererComponent(…) method, 184, 372 getViewport().getView() methods, 363
getWeekdays() method, 93
Global security application

content pane, 17
glass pane, 16–17
JLabel, 15
layered pane, 17
optional MenuBar, 17

Glue components, 56

GraphicsConfiguration object, 267
ScreenLoc class, 268
ScreenPos class, 268
GraphicsEnvironment class, 266
GraphicsEnvironment session, 266
GridLayout Manager
addButtons() method, 59
addComponents() method, 61
application, 58, 60
buttonPress() method, 59
displayConstraints() function, 61
event.getActionCommand() method, 59
GridBagConstraints class, 61–62
layoutContainer() method, 60
pane.getLayout() method, 60
run() method, 59
GroupLayout Manager, 62

headerTask class, 247, 256
Help.help() method, 355
Help.wsadmin() method, 360
Hypertext markup language (HTML)

FormattingVisitor() method, 248 getHeaders class, 246
getHeader scripts, 249
getLinks routine, 232
headerTask class, 247
head(…) method, 248
HTML label, 243
HTML text modification, 244 javadocInfo_03 sample output, 241 Java “HTML” classes, 231
JToggleButton, 245
rendering HTML, 242
textTask class, 241

IBM website, 233
Import(…) event handler method, 458
ImportTask class, 458
Inner cellInfo class, 436
InputMethodListener class, 114–115
InputVerifier method, 227
Interactive scripts

deprecation message, 11
equivalent Java application, 10
Welcome.py script file, 9
Internal frames
iFrameDemo class, 317–319
JDesktopPane class, 322–324
JInternalFrame classes (*see* JInternalFrame classes) layers
labels, 319
LayeredPaneDemo class, 320–321
positioning, 321–322
scratch
application output, 347
consoleTimeout script (*see* consoleTimeout script) InternalFrame class, 343–344
menu items, 341–342
multiple inheritance issue, 344
RadioButton class, 349–351
result, 352
revised consoleTimeout class, 342
setSelected() method, 342
setValue(…) method, 350–351
Internal frames (*cont.*)
stateChange(…) method, 348 TextandButton class, 347–348 TextField class,
344–345 wsadmin script, 336
WSAStask class, 346
Introspection, 4
invokeLater() method, 12
isCellEditable method, 180
isValid() method, 403
isVisible() method, 408

J

java.awt.Dimension class, 199
Javadoc application, 258
JButton class hierarchy

equivalent Jython code, 22
frame.add() method, 23
interactive wsadmin session, 21
Java code, 22
wsadmin interactive session, 22

JColorChooser class
components, 296
constructors, 294–295
sample output, 293–294

JComboBox
cell editors, 175
DynamicComboBox

actionPerformed() method, 89 BorderLayout.CENTER constant, 88 remove() method, 89
run() method, 87–88

editable attributes, 86
event.getSource() method, 85
sample output, 85

JDialog
class hierarchy, 264
CustomDialog1.py sample output, 271 GraphicsConfiguration object, 267
methods, 265

JEditorPanes, 259
JFileChooser class, 454
constructors, 287
dialog types, 291
File Chooser dialog box, 285–286 FileFilter mechanism, 290–291
FileSystemView instance

descendent class, 287–288
restricted filesystem, 288–289 selection types, 292
JFormattedTextField, 166
FormattedTextFieldDemo.py script, 90–91 NumberFormat class, 91
JFrame
class hierarchy, 264
classes function, 24
classInfo() function, 25
classInfo.py, 25
interactive wsadmin session, 27
java.awt.Component.doLayout() description, 26 Java documentation, 23
JFrame “convenience, 29
“layout” attributes, 27
“layout” methods, 26
methods and attribute names, 24–25
wsadmin interactive session, 28
GraphicsConfiguration object, 267
JFrame add() method, 5
JFrame documentation, 19
JFrame Layout Manager, 6
JInternalFrame classes
advantage, 325
events
eventAdapter class, 334–335
eventLogger class, 331–332, 334
iFrameEvents class, 332–333
InternalFrameListener class, 331
run() method, 333
features, 329
findNot() methods, 329
FrameMethod application, 325–326
iFrameEvents2.py script, 335
vs. JFrame, 330
makeMenu() method, 328
run() method, 327–328
showItems() methods, 329
textFile(…) method, 326
utility methods, 326–327
JLabel

adding second label, 5
definition, 4
JFrame class, 5
window, 5
JList
BorderLayout frame, 112
count() method, 109
java.util.Vector, 105–106
JViewport, 107
ListSelectionListener class, 113–114
list selection mode, 110–111
run() method, 108–109
ScrollPane instance, 106–107
text input field
DocumentListener methods, 116
InputMethodListener class, 114–115
iterations, 118
KeyListener events, 116
keyword argument lists, 119
run() method, 116–118
textCheck() method, 119
JMenu entries, 125
JOptionPane methods
JOption show*Dialog method name variants, 274 showConfirmDialog() method, 279
showInputDialog() method, 283
showMessageDialog() method, 274
showOptionDialog() method, 276
JPasswordField
ActionListener event handler method, 82–83 character-obfuscation property, 80–81
event.getActionCommand() method, 82
frame.pack() method, 80
getPassword() method, 82
PasswordDemo class, 79–80
setEchoChar() method, 79
toString() method, 84–85
Jsoup library
javadocInfo_01.py sample output, 237

JTabbedPane, 253
simple Jsoup demonstration, 235
soupTask class, 236
steps, 234
URL, combo box to list box, 238
JSpinner class, 182
JSpinner field
DateFormatSymbols method, 92–93
default spinner constructor, 94
Spinner1 class, 92
spinner editor, 96–97
SpinnerModel argument, 94
SpinnerNumberModel class
calendarField argument, 96
default SpinnerDateModel, 96
Spinner3 class, 95
zero parameter constructor, 95
value selection, 93–94
JSpinner renderer, 181
JSplitPane method, 251, 259
JTabbedPane, 253
JTable class
cell renderers
custom cell renderers, 163
data type rendering, 163
data type-specific cell renderers, 162
columnSelectionAllowed and rowSelectionAllowed attributes, 158
getTableHeader() method, 159
individual cell selection, 158
rowHeight Getter and Setter methods, 177 row selection and editing, 156
selectionMode property, 156
setReorderingAllowed() method, 159
table models
AbstractTableModel, 160
DefaultTableModel, 160
getColumnClass(…) method, 160 isCellEditable() method, 160 read-only
TableModel class, 160
write-only attributes, 157
JTableHeader class, 184

JTree class
DefaultTreeCellEditor, 150–152
DefaultTreeCellRenderer, 152
makeTree() method, 149–150
manipulation
buttonRow() method, 147–148 DynamicTree images, 146
getSuffix() method, 147
setLocationRelativeTo() method, 147
Tree1.py script
branch node, 138
cellTree() method, 139
description, 137
JScrollPane, 138
root node, 138
sample output, 138
structures, 139
TreeSelectionListener interface
event handler, 143–144
line description, 145
run() method, 144
valueChanged() method, 143
TreeSelectionModel class
constants, 141
node selection, limitation, 142 run() method, 140–141
selectmode attributes, 141
setSelectionMode() method, 142 Jython, 3

Keyboard events
ActionListener method, 214 KeyListener descendant class, 211 Listen3 class, 213

KeyListener descendant class, 211 KeyListener methods, 211
Keystrokes

binding (*see* Bindings)
center application, 199
JTable, 197

KeyBindings, 201
locationRelativeTo = None, 198 table data, 200
table properties, 200

Keyword arguments, 206
horizontalAlignment, 168
selectionMode, 157

Label
buttonPressed() method, 33–34 event handler, 33
run() method, 33
text field, 33

layoutContainer() method, 60
Layout Manager's show() method, 43 Listener methods

absolute layout technique, 222 adapter classes, 209
ComponentEventDemo.java, 217 componentResized() method, 221 Frame1 class, 220
Frame1 sample output, 220
frame.getBounds() method, 221 input fields monitoring, 223
InputVerifier, 227
JButton listeners, 206
keyboard events

ActionListener method, 214 KeyListener descendant class, 211 KeyListener methods, 211 Listen3 class, 213
Listen3.py sample output, 212

Listen1.py application, 208
Listen4 class, 215
MouseListener methods, 208
PropertyChangeListener, 224

List filtering, 252
ListPorts.py script file, 153
ListSelectionListener class, 113–114 ListSelectionListener event handler, 240
listServerPorts(…) method, 421

Mnemonics, 131
Modified cellTree() method, 430 MouseListener methods, 208

P, Q

pane.getLayout() method, 60
parseMethodHelp(…) method, 367
pickATcmd(…) method, 384
pickATgroup(…) method, 384
PortLookupTask class, 438
PortTableModel class, 439, 446
Progress bar class

constructors, 299
doInBackground(…) method, 305 ProgressBar0.py, 300–301
propertyUpdate() method, 304
sample images, 299
stringPainted property, 303–304
SwingWorker class

output, 301
ProgressBar2.py script, 302–303

ProgressMonitor class
constructors, 306
interactive session, 307
isCanceled() method, 310–311
JDialog instances, 307–308
message parameter, 311
parentComponent argument, 313 ProgressMonitor1.py, 308–309
ProgressMonitorInputStream objects, 315 properties, 313–314
propertyUpdate() method, 309
setNote(…) method, 312–313

PropertyChangeListener event handler method, 224, 250

M, N, O

MenuBar(…) method, 442
Menus
accelerator, 132

actionPerformed() method, 126 addActionListen() method, 126 adding
MenuBar, 122
check boxes, 129
class hierarchy, 121
contrasting menu entries, 125
empty JMenuBar, 123
foreground and background colors, 124 MenuBar methods, 122
mnemonics, 131
pop-up menu, 134
radio buttons, 127
Merriam Webster page, 233
MessageType constants, 275

readFileSAX routine, 450
Reflection, 4
removeChoosableFileFilter() method, 291 reportConfiguredPorts(…) method,
422 reportTableModel, 408
Rigid area creation method, 57
Row filtering, 406
rowFinder class, 403
run() method, 32, 59, 404

SaveTask and DiscardTask classes, 445 SAXhandler methods, 450
ScreenLoc class, 268
ScreenPos class, 268
Scripting report method, 387
Security configuration report

administration console, 385
AdminTask method, 416
code changes, 409
column widths

output, 399
processReport routine, 397 setColumnWidths() method, 397 WebSphere

Application Server, 397

frame resize listener, 400
modifications, 390
quick and dirty attempt, 389
rowFinder class, 403
scripting report method, 387
section visibility

clicker(…) method, 413
Find(…) method, 414
modified clicker(…) method, 415 reportTableModel, 412
scopeName, 411
sectionFilter class, 412
testing, 415

table model and cell renderer
HTML coloring, 394
multiple selection issue, 392 revised cell renderer, 395
script, 392

table row filtering, 406
upDownAction class, 404

Selectable input components
check boxes, 100–101
radio buttons, 101–102
toggle buttons

application window, 99
ButtonGroupDemo, 103
getItem() method, 100
JToggleButton constructor, 99

setBounds() method, 36
setColumnWidths() method, 397, 441 setDividerLocation(…) methods, 251
setDividerSize() method, 47
setFileSelectionMode() method, 292 setHighlighter(…) method, 362
setProgress(…) method, 314
setRightComponent(…) method, 259 setSelectionMode() method, 111

setText(…) method, 251
setValidating(…) method, 452
setValueAt(…) method, 172, 174
setVisible() method, 408
showConfirmDialog() method, 279
showDialog() method, 291
showInputDialog() method, 283
showMessageDialog() method, 274
showOpenDialog() method, 291
showOptionDialog() method, 276
showSaveDialog() method, 291
SimpleDialog class, 270
SplitPane PropertyChangeListener output, 250 Split panes

application, 44
component size
attributes, 46
and divider bar, 49

horizontal separation, 44
limitation, 44
nested split panes, 47
OneTouchExpandable divider, 48
oneTouchExpandable keyword argument, 48 setDividerSize() method, 47
vertical splits, 44

startElement(…) method, 451, 456

Swing threads
equivalent approach, 13
Runnable class, 12
Swing component-creation operations, 13 SwingUtilities/EventQueue class, 12
template script, 12–13

SwingWorker setProgress(…) method, 302

TabbedPane, 49
Table1.py output, 154

Text input fields

admin console inactivity timeout, 65 AvailableFonts class, 74
consoleTimeout1 class, 67
JTextArea

caretUpdate(), 75
listeners, 75
setEditable() method, 74 SimpleEditor class, 76

JTextField
ActionListener, 66
TextAlignment class, 71–72

SwingWorker class
concurrency, 68
output, 69
subclass, 69
threads, 68

Text input fields (*cont.*)
update() method, 68, 70
WSAStask class, 69–70

wsadmin application, 66
Top-level containers, 1
treeNodesChanged() method, 151 TreeSelectionListener class, 430, 435, 437

U
update() method, 345

valueChanged(…) event handler method, 375 valueChanged(…) method, 374

W, X, Y, Z

WAShome(…) routine, 435
WASprofileName(…) routine, 435
WASvarLookup(…) routine, 435
WASversion(…) routine, 435
WebSphere Application Server, 397
WebSphere Port (WASports) application

AdminConfig methods, 423
administration console, 419
AdminTask.listServerPorts() method, 420
AdminTask.reportConfiguredPorts(…) method, 422 cell and node tree items

cellTSL class, 435
sample output, 432
utility routines, 432

configurations, 461
creation, 425
empty internal frame, 425
export functionality

API (*see* Application Programming
Interface (API))
DTD, 452
ExportTask class, 453
xml.dom.minidom, 448
import functionality, 457
JSplitPane
cell hierarchy tree, 427
internal frame, 426
menu items, 442
report generation, 461
save and discard, 444
serve port number
PortLookupTask class, 439
PortTableModel class, 439
Sample output, 436
TreeSelectionListener class, 437
utility routine, 437
split pane, 428
table column width, 440
table sorting, 461
text highlighting, 461
WebSphere Port(WASports) application WindowAdapter class, 447
WSAShelp application
adding menu, 376

adding split panes, 358
AdminTask.help(‘-commandGroups’) adding menu, 383
AdminTask.help(…) method, 379 AdminTask help text, 383
ATcommandTask class, 379, 381 ATgroupsTask class, 380–381
clusterConfig step, 384
implementation, 382
pickATcmd(…) method, 384
pickATgroup(…) method, 384 showCmdGroups(…) method, 381 split pane, 383
SwingWorker descendent class, 380 text pane selectable, 383
AdminTask.help(‘-commands’)
AdminTask menu item, 378
doInBackground(…) method, 377 output, 377
“Show” menu item, 378
SwingWorker class, 377
cellSelector listener class, 375
displaying methods
camelWords(…) method, 372 DefaultHighlighter class, 367
getFontMetrics(…) method, 371
getTableCellRendererComponent(…) method, 372
highlight table text, 367
installInteractive method, 375
interactive wsadmin session, 371
JTable, 371
JTextPane component, 367
methodTableModel class, 368–369
MethodTable scripts, 368
methRenderer class, 371–372
output, 370
parseMethodHelp(…) method, 366
proof of concept (PoC), 366, 368
run(…) method, 366
setColumnWidths(…) method, 369
setHiText(…) method, 372
setWidths(…) method, 372–373 Help.help() method, 355
ListSelectionListener class, 373
output, 356
scripting object, 374–375
tabbed pane, 357
text highlighting

ActionListener event handler, 361 advantage, 361
ChangeListener method, 363 DefaultHighlighter class, 362
DefaultHighlightPainter class, 361 find(…) method, 362
Help.wsadmin() method, 360
Highlight class, 360
search(…) method, 361–362
setHighlighter(…) method, 362 tabbed highlighting complications, 363
valueChanged(…) event handler
method, 375
valueChanged(…) method, 374

Swing for Jython

Jython UI and Scripts Development using Java Swing and WebSphere Application Server

Robert A. Gibson

Swing for Jython: Graphical Jython UI and Scripts Development using Java Swing and WebSphere Application Server
Copyright © 2015 by Robert Gibson

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 by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4842-0818-2
ISBN-13 (electronic): 978-1-4842-0817-5

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: Welmoed Spahr
Lead Editor: Steve Anglin
Development Editor: Tracy Brown Hamilton
Technical Reviewers: Rohan Walia, Dhrubojyoti Kayal, Manuel Jordan Elera, and Frank Wierzbicki Editorial Board: Steve Anglin, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Michelle Lowman,

James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Steve Weiss
Coordinating Editor: Mark Powers
Copy Editor: Kezia Endsley
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

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.

Apress and friends of ED books 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 Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this

text is available to readers at www.apress.com/9781484208182. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

I thank God for his countless gifts and blessings and dedicate this work to my bride Linda, and our children. Thank you for loving me, and for putting up with me and my fondness of puns. I'm sorry for all the time that this has required, but I have thought of you all throughout its development. I also thank everyone who helped make this book a reality. I could not have done this without your assistance, nor would it have been anywhere as good as you have helped make it.

Contents

About the Author ✿✿✿

Depending “Too Much” on Limited Information
Swing
Threads
Summary
■Chapter 3: Building a Simple Global Security
Application
Adding Text to the Application Using a
JLabel
No “Pane,” No
Gain
When You Live in a Glass House, Everything Is a
Pane
16
The Layered Look Can Also Be a
Pane
17

vii The Optional MenuBar
17
The Content Pane Will Contain Most of the Visible Items
17
Summary

■ Chapter 4: Button Up! Using Buttons and
Labels
JButton Class Hierarchy
The Layout of the
Land
Buttons! Labels! Action!
Updating the Application
Summary

■ Chapter 5: Picking a Layout Manager Can Be a

Pane
The Absolute Layout Manager Does Not Corrupt Absolutely
Going with the Flow: The FlowLayout Manager
South of the Border: The BorderLayout Manager
What’s in the Cards? Using the CardLayout Manager
Splitting Up Is Easy to Do: Using Split Panes
Vertical Splits: Not as Painful as They Sound
44
Limited Resources: Setting Size Attributes
46
Nested Split Panes
47
Divider and Conquer
47
Rules for Using Split Panes
49
Can I Run a Tab? Using a TabbedPane
Are You Boxed In? Using the BoxLayout Manager
The Box Class
53
Building a Box
54
Invisible Box Components
55

Boxes and Resizable
Components
57

Gridlock, Anyone? Using the GridLayout Manager
58
Shaking Things Up: The GridBagLayout
Manager
Looking at Other Layout Managers
Summary
■Chapter 6: Using Text Input
Fields
What Does It Take to Get Data Into an Application?
JTextField: Getting Data Into the
Application
Your First, Almost Real,
Application
Help Me SwingWorker, You’re My Only Hope!
Back to the JTextField
Size Matters: Looking at Text Font Attributes
72
The Elephant (Font) in the Room
Using JTextArea for Input
Summary
■Chapter 7: Other Input
Components
Password Fields
Is There an Echo in Here? Using the Character-Obfuscation

Property
80
The getPassword()
Method
82
The event getActionCommand() Method
82
The JPasswordField Event Handler
82
Converting jarray Values to Strings
84
Choosing from a List
Editing a
ComboBox
86
Using the DynamicComboBox
Formatted Text Fields
Using a JSpinner Text Field
Some DateFormatSymbols Methods
92
The JSpinner Class
93
The SpinnerModel Class
94 The JSpinner Editor
96
Summary
■Chapter 8: Selectable Input
Components

Toggle Buttons
Check Boxes
Radio Buttons
Toggle Buttons in a Button Group
Summary
■Chapter 9: Providing Choices, Making Lists
Making a List and Checking It Twice
Optional Scroll Bars
The ScrollPane Viewport
107 Manipulating the List
Counting List Items
108 Limiting the Selectable Items
110 Reacting to List-Selection Events
111 Reacting to User (Text) Input
114
Summary
■Chapter 10: Menus and MenuItems
The JMenu Class Hierarchy
Reacting to Menu-Related Events
Using Radio Buttons on a Menu
Using Check Boxes on a Menu

Separating Menu Items
Menu Mnemonics and Accelerators
Pop-Up
Menus
Summary

x

■Chapter 11: Using JTree to Show the Forest: Hierarchical Relationships of
Components

Displaying the Servers in a WebSphere Application Server Cell
137
JTree Attributes and Methods
The TreeSelectionModel
Class
140
TreeSelectionListener
143
JTree
Manipulation
The DefaultTreeModel Class
149
The TreeModelListener Interface
150
Summary
■Chapter 12: Motion to Take from the Table: Building
Tables
Tables Can Be Really
Easy
Defaults Can Be Harmful to your Mental
Health
Picky, Picky, Picky Selecting Parts of a

Table
155
Row, Row, Row Your
Table? Working with
Rows
156
Selecting
Columns
158
Selecting Individual
Cells
158
I Am the Very Model of a Modern Major General: Table Models
Types of Table
Models
159
Cell
Renderers
Custom Cell Renderers
163
Using Cell Editors
Boolean Cell
Editors
172
Numeric Cell Editors
173
Custom Numeric Cell
Editors
174
JComboBox Cell
Editors
175
Warning: Ugliness Ahead
178
Column Manipulation

Column Widths 183
Column Heading Width 184
Determining Column Width 185
Column Adjustments 188
Summary
■Chapter 13: Keystrokes, Actions, and Bindings, Oh My!
Getting in a Bind: Looking at Bindings
What Is Meant by Binding? 193
InputMaps and ActionMaps 193
JTable Keystrokes
Putting It All Together
locationRelativeTo = None 198
Centering the Application 199
Defining the Table Properties 200
Computing the Table Data 200
The Fruits of Your Labor 201
Binding Reuse
Where to Begin: Finding the Appropriate Action Class

203 What Do You Need to Worry About? Boundary Conditions
203
Summary
■Chapter 14: It’s the Event of the Year: Events in Swing Applications
If an Event Occurs and No One Hears It
Using Listener Methods
Put Your Listener Where Your Component Is
Adapt or Die: Using Adapter Classes
Listening for Keyboard Events
Most Objects Never Really Listen
Looking for a Listener in a Haystack
Using a ComponentAdapter to Monitor Changes
Monitoring the Input Fields

Using a PropertyChangeListener
224
Using an InputVerifier
Summary
■Chapter 15: Nuts to Soup: Using Jsoup to Enhance Applications
Using Existing Classes: Creating an HTML Retrieval Application from Scratch 231
Wouldn’t It Be Nice: Using Java Libraries

Working with the Jsoup Library
Jsoup Call May Appear to Hang
From a Combo Box to a List Box
Adding a TextArea to Show the HTML
Rendering HTML
Modifying the HTML Text
Identifying the Sections
Fixing the Great Divide
Filtering the List
Using Jsoup to Pick Up the Tab: Adding a JTabbedPane
Adding Tabbed Editor Panes to the Javadoc Application
What Improvements/Enhancements Remain?
Summary
■Chapter 16: Conversing with a User with Dialog Boxes
What Are Dialog Boxes?
What’s a JDialog?
What’s the GraphicsConfiguration Component Do?
Using a GraphicsConfiguration Object
Is a GraphicsConfiguration Object Really Necessary?
What About an Owner?

Where’s the Dialog?
Multiple Modal Dialog Boxes Are
Annoying

Using JOptionPane Methods
274
The JOptionPane showMessageDialog()
Method
274
The JOptionPane showOptionDialog()
Method
276
The JOptionPane showConfirmDialog() Method
279
The JOptionPane showInputDialog()
Method
279
Summary
■Chapter 17: Specialized Dialog
Boxes
The JFileChooser Class
JFileChooser
Constructors
287
Using a FileSystemView
287
File
Filtering
290
Chooser Dialog
Types
291

Selection
Types
292
The JColorChooser
Class
The javax.swing.colorchooser
Package
294
What Else Do You Need to Know About These “Special” Dialog Boxes?
Summary
■Chapter 18: Monitoring and Indicating Progress
Changing the
Cursor
Showing a Progress
Bar
SwingWorker Progress
301
Showing Progress
Details
303
Specifying a Progress Bar Range
304
Indeterminate ProgressBar
Range
305
ProgressMonitor
Objects
ProgressMonitor Cancellation
309
The ProgressMonitor Message
311
The ProgressMonitor
Note

312
The ProgressMonitor
parentComponent
313 Other ProgressMonitor Properties
313 ProgressMonitorInputStream Objects
Summary
■Chapter 19: Internal
Frames
Looking at Inner Frames
Layers
Position Within the Layer
321 The JDesktopPane Class
JFrame or JInternalFrame?
The JInternalFrame
Class
JFrame and JInternalFrame
Methods
325 More JFrame and JInternalFrame Differences
330 JInternalFrame
Events
331 More JInternalFrame Topics
335 Building an Application from
Scratch
Simple Non-GUI Jython Version of the consoleTimeout Script
336 First GUI Jython Version of the consoleTimeout
Script
337 Adding Menu
Items
341 Changing from JFrame to
JInternalFrame

342
Summary
■Chapter 20: Building a Graphical Help Application
Showing the Help Text
Using a Tabbed Pane
Adding Split Panes
Text Highlighting
Adding Text Highlighting to WSAShelp Application
362 Tabbed Highlighting Complications
363

Displaying Methods in a Table
365
Highlighting Text Within the Table
367
Using Tables in the Help Application
368
Fixing the Table Appearance
371
Selecting Table Cells
Adding a Menu
AdminTask help(‘-commands’)
How Do We Find and Identify Existing Commands?

377
Showing the User that Something Is Happening
377
AdminTask help('-commandGroups')
Another ATgroupsTask Implementation
382
Step It Up: Displaying the "Steps" Help Text
How Should You Do It?
383
Should You Add a Menu?
383
Should You Add Another Split Pane?
383
Can You Make Parts of the Text Pane Selectable?
383
And Now for Something Completely Different
384
Summary
■Chapter 21: A Security Configuration Report
Application
Generating the Administration Console Report
The Scripting Report Method
First
Attempt
Second Attempt, Ignoring the Last
Delimiter
Adding a Table Model and Cell

Renderer
Using Color Instead of a Bold Font
394
Adjusting Column Widths
Column Widths and Row
Heights
397
Adding a Frame Resize
Listener
Fixing the Row Selection Colors
402

Which Rows Are Visible?
403
Table Alignment in the
Viewport
Table Row
Filtering
Finding Text
Section Visibility
Does It Work?
414
Progress Indicator
Summary
■Chapter 22: WASports: A WebSphere Port
Application
Using the Administration Console
The AdminTask listServerPorts( )
Method
The AdminTask reportConfiguredPorts( ) Method

Using AdminConfig Methods
Step 0: Creating a WASports Application
Step 1: Adding an Empty Internal Frame
Step 2: Adding an Empty JSplitPane to the Internal Frame
Step 3: Adding a Cell Hierarchy Tree to the JSplitPane
Step 4: Updating the Right Pane
Step 5: Displaying Cell and Node Details
Utility Routines
432
New Classes for WASports_05
435
Step 6: Displaying Server Port Number Information
Step 7: Computing Table Column Widths
Step 8: Adding Menu Items
Step 9: Implementing Save and Discard
Step 10: Implementing the Export Functionality
Using the Document Object Model API
448
Using the Simple API for XML (SAX)
449

Initial WASports DTD 452
The ExportTask Class 453
Step 11: Implementing the Import Functionality 458
The ImportTask Class 458
What's Left? 461
Text Highlighting 461
Table Sorting 461
Comparing Configurations 461
Report Generation 461
Summary xviii
Index

About the Author

Robert A. (Bob) Gibson is an Advisory Software Engineer with decades of experience in numerous software-related roles at IBM, including Architect, Developer, Tester, Instructor, and Technical Support. While providing technical support for the IBM's WebSphere Application Server product, he was the primary author for "WebSphere Application Server Administration Using

Jython” which was published by IBM Press. He is currently a member of the IBM technical support team responsible for the IBM MQ product on distributed platforms. He holds both a Bachelor of Science degree in Engineering Science and a Master of Science degree in Computer Science from the University of Virginia.

About the Technical Reviewers

Manuel Jordan Elera is an autodidactic developer and researcher who enjoys learning new technologies for his own experiments and creating new integrations.

Manuel won the 2010 Springy Award – Community Champion and Spring

Champion 2013. In his little free time, he reads the Bible and composes music on his guitar. Manuel is known as dr_pompeii. He has tech reviewed numerous books for Apress, including *Pro Spring, 4th Edition* (2014), *Practical Spring LDAP* (2013), *Pro JPA 2, Second Edition* (2013), and *Pro Spring Security* (2013).

Read his 13 detailed tutorials about many Spring technologies, contact him through his blog at http://www.manueljordanelera.blogspot.com, and follow him on his Twitter account, @dr_pompeii

Dhrubojyoti Kayal is a hands-on Java developer and architect. He is an open source evangelist. He has been helping enterprises solve integration challenges and build complex large applications leveraging Java technologies for the past 14 years. His current area of focus is data migration and real-time analytics with Java. Dhrubojyoti is also the author of *Pro Java EE Spring Pattern* (2008) from Apress.

Rohan Walia is a Senior Software Consultant with extensive experience in client/ server, web-based, and enterprise application development. He is an Oracle Certified ADF Implementation Specialist and a Sun Certified Java Programmer. Rohan is responsible for designing and developing end-to-end applications consisting of various cutting-edge frameworks and utilities. His areas of expertise are Oracle ADF, Oracle WebCenter, Fusion, Spring, Hibernate, and Java/J2EE. When not working, Rohan loves to play tennis, hike, and travel. Rohan would like to thank his wife, Deepika Walia, for using all her experience and expertise when reviewing this book.

Frank Wierzbicki is the head of the Jython project and a member of the Python

Software Foundation. He has over 15 years of experience as a software developer, primarily working in Python and Java. He has been programming since the Commodore 64 was the king of home computers (look it up kids!) and can’t imagine why anyone would do anything else for a living. Frank’s most enduring hobby is picking up new programming languages, but he has yet to find one that is more fun to work with than Python.

Python Data Visualization Cookbook

Over 60 recipes that will enable you to learn how to create attractive visualizations using Python's most popular libraries

Igor Milovanović BIRMINGHAM - MUMBAI

Python Data Visualization Cookbook

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: November 2013
Production Reference: 1191113

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-336-7
www.packtpub.com
Cover Image by Gorkee Bhardwaj (afterglowpictures@gmail.com)

Credits

Author
Igor Milovanović Project Coordinator Rahul Dixit

Reviewers
Tarek Amr
Simeone Franklin Jayesh K. Gupta
Kostiantyn Kucher Kenneth Emeka Odoh

Proofreaders
Amy Johnson Lindsey Thomas

Indexer
Mariammal Chettiyar
Acquisition Editor James Jones
Graphics
Abhinash Sahu
Lead Technical Editor Ankita Shashi
Production Coordinator Shantanu Zagade

Technical Editors Pratik More
Amit Ramadas Ritika Singh

Copy Editors
Brandt D'Mello
Janbal Dharmaraj Deepa Nambiar Kirti Pai
Laxmi Subramanian

Cover Work
Shantanu Zagade

About the Author

Igor Milovanović is an experienced developer with a strong background in Linux system and software engineering. He has skills in building scalable data-driven distributed software-rich systems.

He is an Evangelist for high-quality systems design who holds strong interests in software architecture and development methodologies. He is always persistent on advocating methodologies that promote high-quality software, such as test-

driven development, one-step builds, and continuous integration.

He also possesses solid knowledge of product development. Having field experience and official training, he is capable of transferring knowledge and communication flow from business to developers and vice versa.

I am most grateful to my fiance for letting me spend endless hours on the work instead with her and for being an avid listener to my endless book monologues. I want to also thank my brother for always being my strongest supporter. I am thankful to my parents for letting me develop myself in various ways and become the person I am today.

I could not write this book without enormous energy from open source community that developed Python, matplotlib, and all libraries that we have used in this book. I owe the most to the people behind all these projects. Thank you.

About the Reviewers

Tarek Amr achieved his postgraduate degree in Data Mining and Information Retrieval from the University of East Anglia. He has about 10 years' experience in Software Development. He has been volunteering in Global Voices Online (GVO) since 2007, and currently he is the local ambassador of the Open Knowledge Foundation (OKFN) in Egypt. Words such as Open Data, Government 2.0, Data Visualisation, Data Journalism, Machine Learning, and Natural Language Processing are like music to his ears.

Tarek's Twitter handle is @gr33ndata and his homepage is http://tarekamr.appspot.com/.

Jayesh K. Gupta is the Lead Developer of Matlab Toolbox for Biclustering Analysis (MTBA). He is currently an undergraduate student and researcher at IIT Kanpur. His interests lie in the field of pattern recognition. His interests also lie in basic sciences, recognizing them as the means of analyzing patterns in nature. Coming to IIT, he realized how this analysis is being augmented by Machine Learning algorithms with various diverse applications. He believes that augmenting human thought with machine intelligence is one of the best ways to advance human knowledge. He is a long time technophile and a free-software Evangelist. He usually goes by the handle, rejuvyesh online. He is also an avid reader and his books can be checked out at Goodreads. Checkout his projects at

Bitbucket and GitHub. For all links visit http:// home.iitk.ac.in/~jayeshkg/. He can be contacted at a2z.jayesh@gmail.com.

Kostiantyn Kucher was born in Odessa, Ukraine. He received his Master's degree in Computer Science from Odessa National Polytechnic University in 2012. He used Python as well as Matplotlib and PIL for Machine Learning and Image Recognition purposes.

Currently, Kostiantyn is a PhD student in Computer Science specializing in Information Visualization. He conducts his research under the supervision of *Prof. Dr. Andreas Kerren* with the ISOVIS group at the Computer Science Department of Linnaeus University (Växjö, Sweden).

Kenneth Emeka Odoh performs research on state of the art Data Visualization techniques. His research interest includes exploratory search where the users are guided to their search results using visual clues.

Kenneth is proficient in Python programming. He has presented a Python conference talk at Pycon, Finland in 2012 where he spoke about Data Visualization in Django to a packed audience.

He currently works as a Graduate Researcher at the University of Regina, Canada. He is a polyglot with experience in developing applications in C, C++, Python, and Java programming languages.

When Kenneth is not writing source codes, you can find him singing at the Campion College chant choir.

www.PacktPub.com

Support files, eBooks, discount offers and more
You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

TM
http://PacktLib.PacktPub.com Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?
f Fully searchable across every book published by Packt f Copy and paste, print and bookmark content f On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface 1
Chapter 1: Preparing Your Working Environment 5

Introduction 5
Installing matplotlib, NumPy, and SciPy 6
Installing virtualenv and virtualenvwrapper 8
Installing matplotlib on Mac OS X 10
Installing matplotlib on Windows 11
Installing Python Imaging Library (PIL) for image processing 12
Installing a requests module 14
Customizing matplotlib's parameters in code 14
Customizing matplotlib's parameters per project 16
Chapter 2: Knowing Your Data 19
Introduction 19
Importing data from CSV 20
Importing data from Microsoft Excel files 22
Importing data from fixed-width datafiles 25
Importing data from tab-delimited files 27
Importing data from a JSON resource 28
Exporting data to JSON, CSV, and Excel 31
Importing data from a database 36

Cleaning up data from outliers 40
Reading files in chunks 46
Reading streaming data sources 48
Importing image data into NumPy arrays 50
Generating controlled random datasets 56
Smoothing the noise in real-world data 64

Chapter 3: Drawing Your First Plots and Customizing Them 71
Introduction 72
Defining plot types – bar, line, and stacked charts 72
Drawing a simple sine and cosine plot 78
Defining axis lengths and limits 81
Defining plot line styles, properties, and format strings 84
Setting ticks, labels, and grids 89
Adding a legend and annotations 92
Moving spines to the center 95
Making histograms 96
Making bar charts with error bars 99
Making pie charts count 101
Plotting with filled areas 103
Drawing scatter plots with colored markers 105
Chapter 4: More Plots and Customizations 109
Introduction 109
Setting the transparency and size of axis labels 110
Adding a shadow to the chart line 113
Adding a data table to the figure 116
Using subplots 118
Customizing grids 121
Creating contour plots 125
Filling an under-plot area 128
Drawing polar plots 131
Visualizing the filesystem tree using a polar bar 134
Chapter 5: Making 3D Visualizations 139
Introduction 139
Creating 3D bars 139
Creating 3D histograms 143
Animating in matplotlib 146
Animating with OpenGL 150

Chapter 6: Plotting Charts with Images and Maps 157
Introduction 157
Processing images with PIL 158
Plotting with images 164
Displaying an image with other plots in the figure 168
Plotting data on a map using Basemap 172
Plotting data on a map using Google Map API 177
Generating CAPTCHA images 183
Chapter 7: Using Right Plots to Understand Data 189
Introduction 189
Understanding logarithmic plots 190
Understanding spectrograms 193
Creating a stem plot 198
Drawing streamlines of vector flow 201
Using colormaps 205
Using scatter plots and histograms 210
Plotting the cross-correlation between two variables 217
Importance of autocorrelation 220
Chapter 8: More on matplotlib Gems 225
Introduction 225
Drawing barbs 225
Making a box and a whisker plot 229
Making Gantt charts 232
Making errorbars 237
Making use of text and font properties 240
Rendering text with LaTeX 246
Understanding the difference between pyplot and OO API 250
Index 257

The best data is the data that we can see and understand. As developers, we want to create and build the most comprehensive and understandable visualizations. It is not always simple; we need to find the data, read it, clean it, massage it, and then use the right tool to visualize it. This book explains the process of how to read, clean, and visualize the data into information with straight and simple (and not so simple) recipes.

How to read local data, remote data, CSV, JSON, and data from relational databases are all explained in this book.

Some simple plots can be plotted with a simple one-liner in Python using matplotlib, but doing more advanced charting requires knowledge of more than just Python. We need to understand the information theory and human perception aesthetics to produce the most appealing visualizations.

This book will explain some practices behind plotting with matplotlib in Python, statistics used, and usage examples for different charting features we should use in an optimal way. This book is written and the code is developed on Ubuntu 12.03 using Python 2.7, IPython 0.13.2, virtualenv 1.9.1, matplotlib 1.2.1, NumPy 1.7.1, and SciPy 0.11.0.

What this book covers

Chapter 1, *Preparing Your Working Environment*, covers a set of installation recipes and advices on how to install the required Python packages and libraries on your platform. *Chapter 2*, *Knowing Your Data*, introduces you to common data formats and how to read and write them, be it CSV, JSON, XSL, or relational databases.

Chapter 3, *Drawing Your First Plots and Customizing Them*, starts with drawing simple plots and covers some of the customization.

Chapter 4 , *More Plots and Customizations*, follows up from previous chapter and covers more advanced charts and grid customization.

Chapter 5, *Making 3D Visualizations*, covers three-dimensional data visualizations such as 3D bars, 3D histograms, and also matplotlib animations.

Chapter 6, *Plotting Charts with Images and Maps*, covers image processing, projecting data onto maps, and creating CAPTCHA test images.

Chapter 7, *Using Right Plots to Understand Data*, covers explanations and recipes on some more advanced plotting techniques such as spectrograms and correlations. *Chapter 8*, *More on matplotlib Gems*, covers a set of charts such as Gantt charts, box plots, and whisker plots, and also explains how to use LaTeX for rendering text in matplotlib.

What you need for this book

For this book, you will need Python 2.7.3 or a later version installed on your operating system. This book was written using Ubuntu 12.03's Python default version (2.7.3).

Other software packages used in this book are IPython, which is an interactive Python environment that is very powerful, and flexible. This can be installed using package managers for Linux-based OSes or prepared installers for Windows and Mac OSes.

If you are new to Python installation and software installation in general, it is very much recommended to use prepackaged scientific Python distributions such as Anaconda, Enthought Python Distribution, or Python(X,Y).

Other required software mainly comprises of Python packages that are all installed using the Python installation manager, pip, which itself is installed using Python's easy_install setup tool.

Who this book is for

Python Data Visualization Cookbook is for developers who already know about Python programming in general. If you have heard about data visualization but don't know where to start, this book will guide you from the start and help you understand data, data formats, data visualization, and how to use Python to visualize data.

You will need to know some general programming concepts, and any kind of programming experience will be helpful. However, the code in this book is explained almost line by line. You don't need math for this book; every concept that is introduced is thoroughly explained in plain English, and references are available for further interest in the topic.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning. Code words in text are shown as follows: "We packed our little demo in class DemoPIL, so that we can extend it easily, while sharing the common code around the demo function, run_fixed_filters_demo."

A block of code is set as follows: def _load_image(self, imfile): self.im = mplimage.imread(imfile) When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

tidy up tick labels size

```
all_axes = plt.gcf().axes
for ax in all_axes:

for ticklabel in ax.get_xticklabels() + ax.get_yticklabels():
ticklabel.set_fontsize(10)
```

Any command-line input or output is written as follows:

```
$ sudo python setup.py install
```

New terms and important words are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "We then set up a label for the stem plot and the position of baseline, which defaults to 0."

Warnings or important notes appear in a box like this.
Tips and tricks appear like this.
Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.
If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support
Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.
Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the errata submission form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material. We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Preparing Your Working Environment

In this chapter, we will cover the following recipes:
f Installing matplotlib, NumPy, and SciPy

f Installing virtualenv and virtualenvwrapper
f Installing matplotlib on Mac OS X
f Installing matplotlib on Windows
f Installing Python Imaging Library (PIL) for image processing
f Installing a requests module
f Customizing matplotlib's parameters in code
f Customizing matplotlib's parameters per project

Introduction

This chapter introduces the reader to the essential tooling and installation and configuration of them. This is a necessary work and common base for the rest of the book. If you have never used Python for data and image processing and visualization, it is advised not to skip this chapter. Even if you do skip it, you can always return to this chapter in case you need to install some supporting tool or verify what version you need to support the current solution.

Installing matplotlib, NumPy, and SciPy

This chapter describes several ways of installing matplotlib and required dependencies under Linux.

Getting ready

We assume that you already have Linux (preferably Debian/Ubuntu or RedHat/SciLinux) installed and Python installed on it. Usually, Python is already installed on the mentioned Linux distributions and, if not, it is easily installable through standard means. We assume that Python 2.7+ Version is installed on your workstation.

Almost all code should work with Python 3.3+ Versions, but because most operating systems still deliver Python 2.7 (some even Python 2.6) we decided to write the Python 2.7 Version code. The differences are small, mainly in version of packages and some code (xrange should be substituted with range in Python 3.3+).

We also assume that you know how to use your OS package manager in order to install software packages and know how to use a terminal.
Build requirements must be satisfied before matplotlib can be built. matplotlib requires NumPy, libpng, and freetype as build dependencies. In order to be able to build matplotlib from source, we must have installed NumPy. Here's how to do it: Install NumPy (at least 1.4+, or 1.5+ if you want to use it with Python 3) from http://www.numpy.org/.

NumPy will provide us with data structures and mathematical functions for using it with large datasets. Python's default data structures such as tuples, lists, or dictionaries are great for insertions, deletions, and concatenation. NumPy's data structures support "vectorized" operations and are very efficient for use and for executions. They are implemented with Big Data in mind and rely on C implementations that allow efficient execution time.
SciPy, building on top of NumPy, is the de facto standard's scientific and numeric toolkit for Python comprising great selection of special functions and algorithms, most of them actually implemented in C and Fortran, coming from the well-known Netlib repository (see http://www.netlib.org). Perform the following steps for installing NumPy:

1. Install Python-NumPy package:
$ sudo apt-get install python-numpy 2. Check the installed version:
$ python -c 'import numpy; print numpy.__version__'

3. Install the required libraries: • libpng 1.2: PNG files support (requires zlib)
• freetype 1.4+: True type font support

$ sudo apt-get install build-dep python-matplotlib If you are using RedHat or variation of this distribution (Fedora, SciLinux, or CentOS) you can use yum to perform same installation:
$ su -c 'yum-builddep python-matplotlib'
How to do it...

There are many ways one can install matplotlib and its dependencies: from source, from precompiled binaries, from OS package manager, and with prepackaged python distributions with built-in matplotlib.

Most probably the easiest way is to use your distribution's package manager. For Ubuntu that should be:
 # in your terminal, type:
$ sudo apt-get install python-numpy python-matplotlib python-scipy

If you want to be on the bleeding edge, the best option is to install from source. This path comprises a few steps: Get the source, build requirements, and configure, compile, and install.

Download the latest source from code host www.github.com by following these steps:
$ cd ~/Downloads/
$ wget
https://github.com/downloads/matplotlib/matplotlib/matplotlib1.2.0.tar.gz

$ tar xzf matplotlib-1.2.0.tar.gz
$ cd matplotlib-1.2.0
$ python setup.py build
$ sudo python setup.py install

Downloading the example code
You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub. com/support and register to have the files e-mailed directly to you.

How it works...

We use standard Python Distribution Utilities, known as Distutils, to install matplotlib from source code. This procedure requires us to previously install dependencies, as we already explained in the *Getting ready* section of this recipe. The dependencies are installed using the standard Linux packaging tools.

There's more...

There are more optional packages that you might want to install depending on what your data visualization projects are about.

No matter what project you are working on, we recommend installing IPython—an Interactive Python shell that supports PyLab mode where you already have matplotlib and related packages, such as NumPy and SciPy, imported and ready to play with! Please refer to IPython's official site on how to install it and use it—it is, though, very straightforward.

Installing virtualenv and virtualenvwrapper

If you are working on many projects simultaneously, or even just switching between them frequently, you'll find that having everything installed system-wide is not the best option and can bring problems in future on different systems (production) where you want to run your software. This is not a good time to find out that you are missing a certain package or have versioning conflicts between packages that are already installed on production system; hence, virtualenv.

virtualenv is an open source project started by *Ian Bicking* that enables a developer to isolate working environments per project, for easier maintenance of different package versions.

For example, you inherited legacy Django website based on Django 1.1 and Python 2.3, but at the same time you are working on a new project that must be written in Python 2.6. This is my usual case—having more than one required Python version (and related packages) depending on the project I am working on.
virtualenv enables me to easily switch to different environments and have the same package easily reproduced if I need to switch to another machine or to deploy software to a production server (or to a client's workstation).

Getting ready

To install virtualenv, you must have workable installation of Python and pip. Pip is a tool for installing and managing Python packages, and it is a replacement for easy install. We will use pip through most of this book for package management. Pip is easily installed, as root executes the following line in your terminal:

 # easy_install pip

virtualenv by itself is really useful, but with the help of virtualenvwrapper, all

this becomes easy to do and also easy to organize many virtual environments. See all the features at http://virtualenvwrapper.readthedocs.org/en/latest/#features.

How to do it...

By performing the following steps you can install the virtualenv and virtualenvwrapper tools: 1. Install virtualenv and virtualenvwrapper:

$ sudo pip virtualenv
$ sudo pip virtualenvwrapper
 # Create folder to hold all our virtual environments and export the path to it.
$ export VIRTENV=~/.virtualenvs
$ mkdir -p $VIRTENV
 # We source (ie. execute) shell script to activate the wrappers $ source /usr/local/bin/virtualenvwrapper.sh
 # And create our first virtual environment
$ mkvirtualenv virt1

2. You can now install our favorite package inside virt1: **(virt1)user1:~$ pip install matplotlib** 3. You will probably want to add the following line to your ~/.bashrc file: source /usr/loca/bin/virtualenvwrapper.sh
Few useful and most frequently used commands are as follows:
f mkvirtualenv ENV: This creates virtual environment with name ENV and activates it
f workon ENV: This activates the previously created ENV
f deactivate: This gets us out of the current virtual environment

Installing matplotlib on Mac OS X

The easiest way to get matplotlib on Mac OS X is to use prepackaged python distributions such as Enthought Python Distribution (EPD). Just go to the EPD site and download and install the latest stable version for your OS.

In case you are not satisfied with EPD or cannot use it for other reasons such as versions distributed with it, there is a manual (read: harder) way of installing Python, matplotlib, and its dependencies.

Getting ready

We will use the Homebrew project that eases installation of all software that Apple did not install on your OS, including Python and matplotlib. Under the hood, Homebrew is a set of Ruby and Git that automate download and installation. Following these instructions should get the installation working. First, we will install Homebrew, and then Python, followed by tools such as virtualenv, then dependencies for matplotlib (NumPy and SciPy), and finally matplotlib. Hold on, here we go.

How to do it...

1. In your Terminal paste and execute the following command:
ruby <(curl -fsSkL raw.github.com/mxcl/homebrew/go)
After the command finishes, try running brew update or brew doctor to verify that installation is working properly.

2. Next, add the Homebrew directory to your system path, so the packages you install using Homebrew have greater priority than other versions. Open ~/.bash_profile (or /Users/[your-user-name]/.bash_profile) and add the following line to the end of file:
export PATH=/usr/local/bin:$PATH

3. You will need to restart the terminal so it picks a new path. Installing Python is as easy as firing up another one-liner:
brew install python --framework --universal

This will also install any prerequisites required by Python.
4. Now, you need to update your path (add to the same line):
export PATH=/usr/local/share/python:/usr/local/bin:$PATH

5. To verify that installation worked, type python --version at the command line, you should see 2.7.3 as the version number in the response.
6. You should have pip installed by now. In case it is not installed, use easy_install to add pip:
$ easy_install pip

7. Now, it's easy to install any required package; for example, virtualenv and virtualenvwrapper are useful:
pip install virtualenv

pip install virtualenvwrapper

8. Next step is what we really wanted to do all along—install matplotlib: **pip install numpy**
brew install gfortran
pip install scipy

Mountain Lion users will need to install the development version of SciPy (0.11) by executing the following line:
pip install -e git+https://github.com/scipy/ scipy#egg=scipy-dev

9. Verify that everything is working. Call Python and execute the following commands: **import numpy**
print numpy.__version__
import scipy
print scipy.__version__
quit()

10. Install matplotlib:
pip install matplotlib

Installing matplotlib on Windows

In this recipe, we will demonstrate how to install Python and start working with matplotlib installation. We assume Python was not previously installed.

Getting ready

There are two ways of installing matplotlib on Windows. The easier way is by installing prepackaged Python environments such as EPD, Anaconda and Python(x,y). This is the suggested way to install Python, especially for beginners.
The second way is to install everything using binaries of precompiled matplotlib and required dependencies. This is more difficult as you have to be careful about the versions of NumPy and SciPy you are installing, as not every version is compatible with the latest version of matplotlib binaries. The advantage in this is that you can even compile your particular versions of matplotlib or any library as to have the latest features, even if they are not provided by authors.

How to do it...

The suggested way of installing free or commercial Python scientific distributions is as easy as following the steps provided on the project's website.

If you just want to start using matplotlib and don't want to be bothered with Python versions and dependencies, you may want to consider using the Enthought Python Distribution (EPD). EPD contains prepackaged libraries required to work with matplotlib and all the required dependencies (SciPy, NumPy, IPython, and more).

As usual, we download Windows Installer (*.exe) that will install all the code we need to start using matplotlib and all recipes from this book.

There is also a free scientific project Python(x,y) (http://code.google.com/p/pythonxy/) for Windows 32-bit system that contains all dependencies resolved, and is an easy (and free!) way of installing matplotlib on Windows. Because Python(x,y) is compatible with Python modules installers, it can be easily extended with other Python libraries. No Python installation should be present on the system before installing Python(x,y).

Let me shortly explain how we would install matplotlib using precompiled Python, NumPy, SciPy, and matplotlib binaries. First, we download and install standard Python using official MSI Installer for our platform (x86 or x86-64). After that, download official binaries for NumPy and SciPy and install them first. When you are sure that NumPy and SciPy are properly installed, then we download the latest stable release binary for matplotlib and install it by following the official instructions.

There's more...

Note that many examples are not included in the Windows installer. If you want to try the demos, download the matplotlib source and look in the examples subdirectory.

Installing Python Imaging Library (PIL) for image processing

Python Imaging Library (PIL) enables image processing using Python, has an extensive file format support, and is powerful enough for image processing. Some popular features of PIL are fast access to data, point operations, filtering, image resizing, rotation, and arbitrary affine transforms. For example, the histogram method allows us to get statistics about the images.

PIL can also be used for other purposes, such as batch processing, image archiving, creating thumbnails, conversion between image formats, and printing

images.
PIL reads a large number of formats, while write support is (intentionally) restricted to the most commonly used interchange and presentation formats.

How to do it...

The easiest and most recommended way is to use your platform's package managers. For Debian/Ubuntu use the following commands:

$ sudo apt-get build-dep python-imaging
$ sudo pip install http://effbot.org/downloads/Imaging-1.1.7.tar.gz

How it works...

This way we are satisfying all build dependencies using apt-get system but also installing the latest stable release of PIL. Some older versions of Ubuntu usually don't provide the latest releases.

On RedHat/SciLinux:

 # yum install python-imaging
 # yum install freetype-devel
 # pip install PIL

There's more...

There is a good online handbook, specifically, for PIL. You can read it at http://www.pythonware.com/library/pil/handbook/index.htm, or download the PDF version from http://www.pythonware.com/media/data/pil-handbook.pdf.

There is also a PIL fork, Pillow, whose main aim is to fix installation issues. Pillow can be found at http://pypi.python.org/pypi/Pillow and it is easy to install. On Windows, PIL can also be installed using a binary installation file. Install PIL in your Python site-packages by executing .exe from http://www.pythonware.com/products/pil/. Now, if you want PIL used in virtual environment, manually copy the PIL.pth file and the PIL directory at C:\Python27\Lib\site-packages to your virtualenv site-packages directory.

Installing a requests module

Most of the data that we need now is available over HTTP or similar protocol, so we need something to get it. Python library requests makes that job easy.
Even though Python comes with the urllib2 module for work with remote

resources and supporting HTTP capabilities, it requires a lot of work to get the basic tasks done.

Requests module brings new API that makes the use of web services seamless and pain free. Lot of the HTTP 1.1 stuff is hidden away and exposed only if you need it to behave differently than default.

How to do it...

Using pip is the best way to install requests. Use the following command for the same: **$ pip install requests**

That's it. This can also be done inside your virtualenv if you don't need requests for every project or want to support different requests versions for each project. Just to get you ahead quickly, here's a small example on how to use requests:

```
import requests
r = requests.get('http://github.com/timeline.json')
print r.content
```

How it works...

We sent the GET HTTP request to a URI at www.github.com that returns a JSON-formatted timeline of activity on GitHub (you can see HTML version of that timeline at https://github.com/timeline). After response is successfully read, the r object contains content and other properties of the response (response code, cookies set, header metadata, even the request we sent in order to get this response).

Customizing matplotlib's parameters in code

The Library we will use the most throughout this book is matplotlib; it provides the plotting capabilities. Default values for most properties are already set inside the configuration file for matplotlib, called.rcfile. This recipe describes how to modify matplotlib properties from our application code.

Getting ready

As we already said, matplotlib configuration is read from a configuration file. This file provides a place to set up permanent default values for certain

matplotlib properties, well, for almost everything in matplotlib.

How to do it...

There are two ways to change parameters during code execution: using the dictionary of parameters (rcParams) or calling the matplotlib.rc() command. The former enables us to load already existing dictionary into rcParams, while the latter enables a call to a function using tuple of keyword arguments.

If we want to restore the dynamically changed parameters, we can use matplotlib. rcdefaults() call to restore the standard matplotlib settings.

The following two code samples illustrate previously explained behaviors:
Example for matplotlib.rcParams:

```
import matplotlib as mp
mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['lines.color'] = 'r'
```

Example for the matplotlib.rc() call:

```
import matplotlib as mpl
mpl.rc('lines', linewidth=2, color='r')
```

Both examples are semantically the same. In the second sample, we define that all subsequent plots will have lines with line width of 2 points. The last statement of the previous code defines that the color of every line following this statement will be red, unless we override it by local settings. See the following example:

```
import matplotlib.pyplot as plt
import numpy as np
t = np.arange(0.0, 1.0, 0.01)

s = np.sin(2 * np.pi * t)
# make line red
plt.rcParams['lines.color'] = 'r' plt.plot(t,s)

c = np.cos(2 * np.pi * t) # make line thick
plt.rcParams['lines.linewidth'] = '3 plt.plot(t,c)
```

```
plt.show()
```

How it works...

First, we import matplotlib.pyplot and NumPy to allow us to draw sine and cosine graphs. Before plotting the first graph, we explicitly set line color to red using plt. rcParams['lines.color'] = 'r'.

Next, we go to the second graph (cosine function), and explicitly set line width to 3 points using plt.rcParams['lines.linewidth'] = '3'.
If we want to reset specific settings, we should call matplotlib.rcdefaults().

Customizing matplotlib's parameters per project

This recipe explains where the various configuration files are that matplotlib uses, and why we want to use one or the other. Also, we explain what is in these configuration files.

Getting ready

If you don't want to configure matplotlib as the first step in your code every time you use it (as we did in the previous recipe), this recipe will explain how to have different default configurations of matplotlib for different projects. This way your code will not be cluttered with configuration data and, moreover, you can easily share configuration templates with your co-workers or even among other projects.

How to do it...

If you have a working project that always uses the same settings for certain parameters in matplotlib, you probably don't want to set them every time you want to add a new graph code. Instead, what you want is a permanent file, outside of your code, which sets defaults for matplotlib parameters.

matplotlib supports this via its matplotlibrc configuration file that contains most of the changeable properties of matplotlib.

How it works...

There are three different places where this file can reside and its location defines its usage. They are:

f Current working directory: This is where your code runs from. This is the place to customize matplotlib just for your current directory that might contain your current project code. File is named matplotlibrc.

f Per user .matplotlib/matplotlibrc: This is usually in user's $HOME directory (under Windows, this is your Documents and Settings directory). You can find out where your configuration directory is using the matplotlib.get_configdir() command. Check the next command.

f Per installation configuration file: This is usually in your python site-packages. This is a system-wide configuration, but it will get overwritten every time you reinstall matplotlib; so it is better to use per user configuration file for more persistent customizations. Best usage so far for me was to use this as a default template if I mess up my user's configuration file or if I need fresh configuration to customize for a different project.

The following one-liner will print the location of your configuration directory and can be run from shell.
$ python -c 'import matplotlib as mpl; print mpl.get_configdir()' The configuration file contains settings for:

f axes: Deals with face and edge color, tick sizes, and grid display.
f backend: Sets the target output: TkAgg and GTKAgg.
f figure: Deals with dpi, edge color, figure size, and subplot settings. f font: Looks at font families, font size, and style settings.
f grid: Deals with grid color and line settings.
f legend: Specifies how legends and text inside will be displayed.
f lines: It checks for line (color, style, width, and so on) and markers settings. f patch: Patches are graphical objects that fill 2D space, such as polygons and circles;

set linewidth, color, antialiasing, and so on.
f savefig: There are separate settings for saved figures. For example, to make rendered files with a white background.
f text: This looks for text color, how to interepret text (plain versus latex markup)
and similar.
f verbose: It checks how much information matplotlib gives during runtime: silent,

helpful, debug, and debug-annoying.
f xticks and yticks: These set the color, size, direction, and labelsize for major and
minor ticks for x and y axes.

There's more...

If you are interested in more details for every mentioned setting (and some that we did not mention here), the best place to go is the website of matplotlib project where there is up-to-date API documentation. If it doesn't help, user and development lists are always good places to leave questions. See the back of this book for useful online resources.

2

Knowing Your Data

In this chapter we will cover the following recipes:
f Importing data from CSV

f Importing data from Microsoft Excel files
f Importing data from fixed-width datafiles
f Importing data from tab-delimited files
f Importing data from a JSON resource
f Exporting data to JSON, CSV, and Excel
f Importing data from a database
f Cleaning up data from outliers
f Reading files in chunks
f Reading streaming data sources
f Importing image data into NumPy arrays
f Generating controlled random datasets
f Smoothing the noise in real-world data

Introduction

This chapter covers basics about importing and exporting data from various formats. Also covered are ways of cleaning data, such as normalizing values, adding missing data, live data inspection, and usage of some similar tricks to get data correctly prepared for visualization.

Importing data from CSV

In this recipe we will work with the most common file format that one will encounter in the wild world of data, CSV. It stands for Comma Separated Values, which almost explains all the formatting there is. (There is also a header part of the file, but those values are also comma separated.)

Python has a module called csv that supports reading and writing CSV files in various dialects. Dialects are important because there is no standard CSV and

different applications implement CSV in slightly different ways. A file's dialect is almost always recognizable by the first look into the file.

Getting ready

What we need for this recipe is the CSV file itself. We will use sample CSV data that you can download from ch02-data.csv.
We assume that sample datafiles is in the same folder as the code reading it.

How to do it...

The following code example demonstrates how to import data from a CSV file. We will:

1. Open the ch02-data.csv file for reading.
2. Read the header first.
3. Read the rest of the rows.
4. In case there is an error, raise an exception.
5. After reading everything, print the header and the rest of the rows. import csv

```
filename = 'ch02-data.csv'
data = []
try:
with open(filename) as f:

reader = csv.reader(f)
header = reader.next()
data = [row for row in reader]

except csv.Error as e:
print "Error reading CSV file at line %s: %s" % (reader.line_
num, e)
sys.exit(-1)
if header:
print header
print '==================='

for datarow in data: print datarow
```

How it works...

First, we import the csv module in order to enable access to required methods. Then, we open the file with data using the with compound statement and bind it

to the object f. The context manager with statement releases us of care about the closing resource after we are finished manipulating those resources. It is a very handy way of working with resource-like files because it makes sure that the resource is freed (for example, that the file is closed) after the block of code is executed over it.

Then, we use the csv.reader() method that returns the reader object, allowing us to iterate over all rows of the read file. Every row is just a list of values and is printed inside the loop.

Reading the first row is somewhat different as it is the header of the file and describes the data in each column. This is not mandatory for CSV files and some files don't have headers, but they are a really nice way of providing minimal metadata about datasets. Sometimes though, you will find separate text or even CSV files that are just used as metadata, describing the format and additional data about the data.

The only way to check what the first line looks like is to open the file and visually inspect it (for example, see the first few lines of the file). This can be done efficiently on Linux using bash commands such as head as follows:

$ head some_file.csv

During iteration of data, we save the first row in header, while we add every other row to the data list.

Should any errors occur during reading, csv.reader() will generate an error that we can catch and print the helpful message to the user, in order to help detection of errors.

There's more...

If you want to read about the background and reasoning for the csv module, the PEP-defined document *CSV File API* is available at http://www.python.org/dev/peps/pep-0305/. If we have larger files that we want to load, it's often better to use well-known libraries, such as NumPy's loadtxt(), that cope better with large CSV files.

The basic usage is simple as shown in the following code snippet:

```
import numpy
data = numpy.looadtxt('ch02-data.csv', dtype='string', delimiter=',')
```

Note that we need to define a delimiter to instruct NumPy to separate our data as appropriate. The function numpy.loadtxt() is somewhat faster than the similar

function numpy.genfromtxt(), but the latter can cope better with missing data, and you are able to provide functions to express what is to be done during the processing of certain columns of loaded datafiles.

Currently, in Python 2.7.x, the csv module doesn't support Unicode, and you must explicitly convert the read data into UTF-8 or ASCII printable. The official Python CSV documentation offers good examples on how to resolve data encoding issues.
In Python 3.3 and later versions, Unicode support is default and there are no such issues.

Importing data from Microsoft Excel files

Although Microsoft Excel supports some charting, sometimes you need more flexible and powerful visualization and need to export data from existing spreadsheets into Python for further use.

A common approach to importing data from Excel files is to export data from Excel into CSV-formatted files and use the tools described in the previous recipe to import data using Python from the CSV file. This is a fairly easy process if we have one or two files (and have Microsoft Excel or OpenOffice.org installed), but if we are automating a data pipe for many files (as part of an ongoing data processing effort), we are not in a position to manually convert every Excel file into CSV. So, we need a way to read any Excel file.

Python has decent support for reading and writing Excel files through the project www.python-excel.org. This support is available in the form of different modules for reading and writing, and is platform independent; in other words, we don't have to run on Windows in order to read Excel files.

The Microsoft Excel file format changed over time, and support for different versions is available in different Python libraries. The latest stable version of XLRD is 0.90 at the time of this writing and it has support for reading .xlsx files.

Getting ready

First we need to install the required module. For this example, we will use the module xlrd. We will use pip in our virtual environment.

```
$ mkvirtualenv xlrdexample
(xlrdexample)$ pip install xlrd
```

After successful installation, use the sample file ch02-xlsxdata.xlsx.

How to do it...

The following code example demonstrates how to read a sample dataset from a known Excel file. We will:

1. Open the file workbook.
2. Find the sheet by name.
3. Read the cells using the number of rows (nrows) and columns (ncols).
4. For demonstration purposes, we only print the read dataset.

```
import xlrd

file = 'ch02-xlsxdata.xlsx'
wb = xlrd.open_workbook(filename=file)
ws = wb.sheet_by_name('Sheet1')
dataset = []

for r in xrange(ws.nrows):
col = []
for c in range(ws.ncols):

col.append(ws.cell(r, c).value)
dataset.append(col)
from pprint import pprint
pprint(dataset)
```

How it works...

Let us try to explain the simple object model that xlrd uses. At the top level, we have a workbook (the Python class xlrd.book.Book) that consists of one or more worksheets (xlrd.sheet.Sheet), and every sheet has a cell (xlrd.sheet.Cell) that we can then read the value from.

We load a workbook from a file using open_workbook(), which returns the xlrd.book. Book instance that contains all the information about a workbook, such as sheets. We access sheets using sheet_by_name(); if we need all sheets, we could use sheets(), which returns a list of the xlrd.sheet.Sheet instances. The xlrd.sheet.Sheet class has a number of columns and rows as attributes that we can use to infer ranges for our loop to access every particular cell inside a worksheet using the method cell(). There is an xrld. sheet.Cell class, though it is

not something we want to use directly.

Note that the date is stored as a floating point number and not as a separate data type, but the xlrd module is able to inspect the value and try to infer if the data is in fact a date. So we can inspect the cell type for the cell to get the Python date object. The module xlrd will return xlrd.XL_CELL_DATE as the cell type if the number format string looks like a date. Here is a snippet of code that demonstrates this:

```
from datetime import datetime
from xlrd import open_workbook, xldate_as_tuple
…
cell = sheet.cell(1, 0)
print cell
print cell.value
print cell.ctype
if cell.ctype == xlrd.XL_CELL_DATE:

date_value = xldate_as_tuple(cell.value, book.datemode) print
datetime(*date_value)
```

This field still has issues, so please refer to the official documentation and mailing list in case you require extensive work with dates.

There's more...

A neat feature of xlrd is its ability to load only parts of the file that are required in the memory. There is an on_demand parameter that can be passed the value True while calling open_workbook so that the worksheet will only be loaded when requested. For example:

```
book = open_workbook('large.xls', on_demand=True)
```

We didn't mention writing Excel files in this section, partly because there will be a separate recipe for that and partly because there is a different module for that —xlwt. You will read more about it in the *Exporting data to JSON, CSV, and Excel* recipe in this chapter. If you need specific usage that was not covered with the module and examples explained earlier, here is a list of other Python modules on PyPi that might help you out with spreadsheets: http://pypi.python.org/pypi?:action=browse&c=377.

Importing data from fixed-width datafiles

Logfiles from events and time series datafiles are common sources for data visualizations. Sometimes, we can read them using CSV dialect for tab-separated data, but sometimes they are not separated by any specific character. Instead, fields are of fixed widths and we can infer the format to match and extract data.

One way to approach this is to read a file line by line and then use string manipulation functions to split a string into separate parts. This approach seems straightforward, and if performance is not an issue, should be tried first.

If performance is more important or the file to parse is large (hundreds of megabytes), using the Python module struct (http://docs.python.org/library/struct.html) can speed us up as the module is implemented in C rather than in Python.

Getting ready

As the module struct is part of the Python Standard Library, we don't need to install any additional software to implement this recipe.

How to do it...

We will use a pregenerated dataset with a million rows of fixed-width records. Here's what sample data looks like:

```
…
207152670 3984356804116 9532
427053180 1466959270421 5338
316700885 9726131532544 4920
138359697 3286515244210 7400
476953136 0921567802830 4214
213420370 6459362591178 0546
…
```

This dataset is generated using code that can be found in the repository for this chapter, ch02-generate_f_data.py.

Now we can read the data. We can use the following code sample. We will:

1. Define the datafile to read.

2. Define the mask for how to read the data.

3. Read line by line using the mask to unpack each line into separate data fields.
4. Print each line as separate fields.

```
import struct
import string

datafile = 'ch02-fixed-width-1M.data'

# this is where we define how to # understand line of data from the file
mask='9s14s5s'

with open(datafile, 'r') as f:
for line in f:
fields = struct.Struct(mask).unpack_from(line) print 'fields: ', [field.strip() for field in fields]
```

How it works...

We define our format mask according to what we have previously seen in the datafile. To see the file, we could have used Linux shell commands, such as head or more, or something similar.

String formats are used to define the expected layout of the data to extract. We use format characters to define what type of data we expect. So if the mask is defined as 9s15s5s, we can read that as "a string of width nine characters, followed by a string width of 15 characters, further followed by a string of five characters."

In general, c defines the character (the char type in C) or a string of length 1, s defines a string (the char[] type in C), d defines a float (the double type in C), and so on. The complete table is available on the official Python website at http://docs.python.org/ library/struct.html#format-characters.

We then read the file line by line and extract (the unpack_from method) the line according to the specified format. Because we might have extraneous spaces before (or after) our fields, we use strip() to strip every extracted field.

For unpacking, we used the object-oriented (OO) approach using the struct.Struct class, but we could have as well used the non-object approach where the line would be: fields = struct.unpack_from(mask, line)

The only difference is the usage of pattern. If we are to perform more processing using the same formatting mask, the OO approach saves us from stating that format in every call. Also, it gives us the ability to inherit the struct.Struct class in future, extending or providing additional functionality for specific needs.

Importing data from tab-delimited files

Another very common format of flat datafile is the tab-delimited file. This can also come from an Excel export but can be the output of some custom software we must get our input from.

The good thing is that usually this format can be read in almost the same way as CSV files, as the Python module csv supports so-called dialects that enable us to use the same principles to read variations of similar file formats—one of them being the tab delimited format.

Getting ready

We are already able to read CSV files. If not, please refer the *Importing data from CSV* recipe first.

How to do it...

We will re-use the code from the *Importing data from CSV* recipe, where all we need to change is the dialect we are using.

```
import csv
filename = 'ch02-data.tab'
data = []
try:
with open(filename) as f:
reader = csv.reader(f, dialect=csv.excel_tab)
header = reader.next()
data = [row for row in reader]

except csv.Error as e:
print "Error reading CSV file at line %s: %s" % (reader.line_num,
e)
sys.exit(-1)

if header:
print header
print '===================='
```

```
for datarow in data: print datarow
```

How it works...

The dialect-based approach is very similar to what we already did in the *Importing data from CSV* recipe, except for the one line where we instantiate the csv reader object, giving it the parameter dialect and specifying the 'excel_tab' dialect that we want.

There's more...

A CSV-based approach will not work if the data is "dirty", that is, if there are certain lines not ending with just a new line character but that instead have additional \t (tab) markers. So we need to clean special lines separately before splitting them. The sample "dirty" tab-delimited file can be found in ch02-data-dirty.tab. The following code sample cleans data as it reads it:

```
datafile = 'ch02-data-dirty.tab'

with open(datafile, 'r') as f:
for line in f:
# remove next comment to see line before cleanup # print 'DIRTY: ',
line.split('\t')

# we remove any space in line start or end line = line.strip()
# now we split the line by tab delimiter print line.split('\t')
```

We see also that there is another approach—using the split('\t') function.

The advantage of using the csv module approach over split() sometimes is that we can re-use the same code for reading by just changing the dialect and detecting the dialect with the file extension (.csv and .tab) or some other method (for example, using the csv.Sniffer class).

Importing data from a JSON resource

This recipe will show us how we can read the JSON data format. Moreover, we will be using a remote resource in this recipe. It will add a tiny level of complexity to the recipe, but it will make it much more useful because, in real

life, we will encounter more remote resources than local.

JavaScript Object Notation (JSON) is widely used as a platform-independent format to exchange data between systems or applications.
A resource, in this context, is anything we can read, be it a file or a URL endpoint (which can be the output of a remote process/program or just a remote static file). In short, we don't care who produced a resource and how; we just need it to be in a known format, such as JSON.

Getting ready

In order to get started with this recipe, we need the requests module installed and importable (in PYTHONPATH) in our virtual environment. We have installed this module in *Chapter 1*, *Preparing Your Working Environment*.

We also need Internet connectivity as we will be reading a remote resource.

How to do it...

The following code sample performs reading and parsing of the recent activities timeline from the GitHub (http://github.com) site. We will perform the following steps for this:

1. Define the GitHub URL to read the JSON format.
2. Get the contents from the URL using the requests module.
3. Read the content as JSON.
4. For each entry in the JSON object, read the URL value for each repository.

```
import requests

url = 'https://github.com/timeline.json'
r = requests.get(url)
json_obj = r.json()
repos = set()
for entry in json_obj:
try:

repos.add(entry['repository']['url'])
except KeyError as e:
print "No key %s. Skipping..." % (e)

from pprint import pprint pprint(repos)
```

How it works...

First, we use the requests module to fetch a remote resource. This is very straightforward as the requests module offers a simple API to define HTTP verbs, so we just need to issue one get() method call. This method retrieves data and request metadata and wraps it in the Response object so we can inspect it. For this recipe, we are only interested in the Response.json() method, which automatically reads content (available at Response. content) and parses it as JSON and loads it into the JSON object.

Now that we have the JSON object, we can process data. In order to do that, we need to understand what data looks like. We can achieve that understanding by opening the JSON resource using our favorite web browser or command-line tool such as wget or curl.

Another way is to fetch data from IPython and inspect it interactively. We can achieve that by running our program from IPython (using %run program_name.py). After execution, we are left with all variables that the program produced. List them all using %who or %whos.

Whatever method we use, we gain knowledge about the structure of the JSON data and the ability to see what parts of that structure we are interested in.

The JSON object is basically just a Python dictionary (or if more complex, a dictionary of dictionaries) and we can access parts of it using a well-known, key-based notation. We get our list of URLs of recently updated repositories referencing entry['repository']['url'].

entry['repository']['url'] matches this section in the actual JSON file:

```
…
"repository" : {
...
"url" : "https://github.com/ipython/ipython", ...

},
…
```

We can now see how nested structures correspond to multidimensional key indexes in Python code.

There's more...

The JSON format (specified by RFC 4627; refer to http://tools.ietf.org/html/rfc4627.html) became very popular recently, as it is more human readable than XML and is also less verbose. Hence, it's lighter in terms of the syntaxes required to transfer data. It is very popular in the web application domain as it is native to JavaScript, the language used for most of today's rich Internet applications.
The Python JSON module has more capabilities than we have displayed here; for example, we could specialize the basic JSONEncoder/JSONDecoder class to transform our Python data into JSON format. The classical example uses this approach to JSON-ify the Python built-in type for complex numbers.

For simple customization, we don't have to subclass the JSONDecoder/JSONEncoder class as some of the parameters can solve our problems.

For example, json.loads() will parse a float as the Python type float, and most of the time it will be right. Sometimes, however, the float value in the JSON file represents a price value, and this is better represented as a decimal. We can instruct the json parser to parse floats as decimal. For example, we have this JSON string:

```
jstring = '{"name":"prod1","price":12.50}'
```

This is followed by these two lines of code:

```
from decimal import Decimal
json.loads(jstring, parse_float=Decimal)
```

The preceding two lines of code will generate this output:

```
{u'name': u'prod1', u'price': Decimal('12.50')}
```

Exporting data to JSON, CSV, and Excel

While, as producers of data visualization, we are mostly using other people's data; importing and reading data are major activities. We do need to write or export data that we produced or processed, whether it is for our or others' current or future use.

We will demonstrate how to use the previously mentioned Python modules to import, export, and write data to various formats such as JSON, CSV, and

XLSX.
For demonstration purposes, we are using the pregenerated dataset from the *Importing data from fixed-width datafiles* recipe.

Getting ready

For the Excel writing part, we will need to install the xlwt module (inside our virtual environment) by executing the following command:

$ pip install xlwt

How to do it...

We will present one code sample that contains all the formats that we want to demonstrate: CSV, JSON, and XLSX. The main part of the program accepts the input and calls appropriate functions to transform data. We will walk through separate sections of code, explaining its purpose.

1. Import the required modules.

```
import os
import sys
import argparse

try: import cStringIO as StringIO
except:
import StringIO
import struct
import json
import csv
```

2. Then, define the appropriate functions for reading and writing data.

```
def import_data(import_file):
'''
Imports data from import_file.
Expects to find fixed width row
Sample row: 161322597 0386544351896 0042
'''
mask = '9s14s5s'
data = []
with open(import_file, 'r') as f:

for line in f:
# unpack line to tuple
```

```
fields = struct.Struct(mask).unpack_from(line) # strip any whitespace for each field
# pack everything in a list and add to full dataset data.append(list([f.strip() for f in fields]))

return data

def write_data(data, export_format):
'''Dispatches call to a specific transformer and returns data
set.
Exception is xlsx where we have to save data in a file. '''
if export_format == 'csv':
return write_csv(data)
elif export_format == 'json':
return write_json(data)
elif export_format == 'xlsx':
return write_xlsx(data)
else:
raise Exception("Illegal format defined")
```

3. We separately specify separate implementation for each data format (CSV, JSON, and XLSX).

```
def write_csv(data):

'''Transforms data into csv. Returns csv as string. '''
# Using this to simulate file IO,
# as csv can only write to files.
f = StringIO.StringIO()
writer = csv.writer(f)
for row in data:

writer.writerow(row)
# Get the content of the file-like object
return f.getvalue()

def write_json(data):
'''Transforms data into json. Very straightforward. '''
j = json.dumps(data)
return j
```

```
def write_xlsx(data):
'''Writes data into xlsx file.

'''
from xlwt import Workbook
book = Workbook()
sheet1 = book.add_sheet("Sheet 1")
row = 0
for line in data:

col = 0
for datum in line:
print datum
sheet1.write(row, col, datum)
col += 1
row += 1
# We have hard limit here of 65535 rows # that we are able to save in
spreadsheet.
if row > 65535:
print >> sys.stderr, "Hit limit of # of rows in one sheet (65535)."
break
# XLS is special case where we have to
# save the file and just return 0
f = StringIO.StringIO()
book.save(f)
return f.getvalue()
```

4. Finally, we have the main code entry point, where we parse argument-like files from the command line to import data and export it to the required format.

```
if __name__ == '__main__':
# parse input arguments
parser = argparse.ArgumentParser()
parser.add_argument("import_file", help="Path to a fixed-width

data file.")
parser.add_argument("export_format", help="Export format:
json, csv, xlsx.")
args = parser.parse_args()
```

```
if args.import_file is None:
print >> sys.stderr, "You myst specify path to import from."
sys.exit(1)
if args.export_format not in ('csv','json','xlsx'): print >> sys.stderr, "You must
provide valid export file format."
sys.exit(1)
# verify given path is accesible file
if not os.path.isfile(args.import_file):
print >> sys.stderr, "Given path is not a file: %s" % args.import_file
sys.exit(1)
# read from formated fixed-width file data = import_data(args.import_file)

# export data to specified format
# to make this Unix-lixe pipe-able # we just print to stdout
print write_data(data, args.export_format)
```

How it works...

In one broad sentence, we import the fixed-width dataset (defined in the *Importing data from fixed-width datafiles* recipe) and then export that to stdout so we can catch that in a file or as input to another program.

We call out the programmer from the command line giving two mandatory arguments: the input filename and the export data format (JSON, CSV, and XLSX).

If we successfully parse those arguments, we dispatch the input file reading to a function import_data(), which returns the Python data structure (list of lists) that we can easily manipulate to get to the appropriate export format.

We route our request inside the write_data() function, where we just forward a call to the appropriate function (write_csv(), for example).
For CSV, we obtain the csv.writer() instance that we use to write every line of data we iterate over.
We just return the given string as we will redirect this output from our program to another program (or just to cat in a file).

The JSON export is not required for this example, as the json module provides us with the dump() method that happily reads our Python structure. Just as for CSV, we simply return and dump this output to stdout.

The Excel export requires more code, as we need to create a more complex model of the Excel workbook and worksheet(s) that will hold the data. This activity is followed by a similar iterative approach. We have two loops, the outer one goes over every line in the source dataset iterated, while the inner one iterates over every field in the given line.

After all this, we save the Book instance into a file-like stream that we can return to stdout and use it both in read files and the files consumed by the web service.

There's more...

This, of course, is just a small set of possible data formats that we could be exporting to. It is fairly easy to modify the behavior. Basically, two places need changes: the import and export functions. The function for import needs to change if we want to import a new kind of data source.

If we want to add a new export format, we need to first add functions that will return a stream of formatted data. Then, we need to update the write_data() function to add the new elif branch to have it call our new write_* function. One thing we could also do is make this a Python package, so we can re-use it over more projects. In that case, we would like to make import more flexible and maybe add some more configuration features for import.

Importing data from a database

Very often, our work on data analysis and visualization is at the consumer end of the data pipeline. We most often use the already produced data, rather than producing the data ourselves. A modern application, for example, holds different datasets inside relational databases (or other databases), and we use these databases and produce beautiful graphs.

This recipe will show you how to use SQL drivers from Python to access data.

We will demonstrate this recipe using a SQLite database because it requires the least effort to setup, but the interface is similar to most other SQL-based database engines (MySQL and PostgreSQL). There are, however, differences in the SQL dialect that those database engines support. This example uses simple SQL language and should be reproducible on most common SQL database engines.

Getting ready

To be able to execute this recipe, we need to install the SQLite library.
$ sudo apt-get install sqlite3

Python support for SQLite is there by default, so we don't need to install anything Python related. Just fire the following code snippet in IPython to verify that everything is there:

```
import sqlite3
sqlite3.version
sqlite3.sqlite_version
```

We get the output similar to this:
In [1]: import sqlite3
In [2]: sqlite3.version Out[2]: '2.6.0'
In [3]: sqlite3.sqlite_version Out[3]: '3.6.22'
Here, sqlite3.version gets us the version of the Python sqlite3 module, and sqlite_version returns the system SQLite library version.

How to do it...

To be able to read from the database, we need to:

1. Connect to the database engine (or the file in the case of SQLite).
2. Run the query against the selected tables.
3. Read the result returned from the database engine.

I will not try to teach SQL here, as there are many books on that particular topic. But just for clarity, we will explain the SQL query in this code sample:
SELECT ID, Name, Population FROM City ORDER BY Population DESC LIMIT 1000

ID , Name, and Population are columns (fields) of the table City from which we select data. ORDER BY tells the database engine to sort our data by the Population column, and DESC means descending order. LIMIT allows us to get just the first 1,000 records found.

For this example, we will use the world.sql example table, which holds the world's city names and populations. This table has more than 5,000 entries. This is how our table looks:

```
 1        ID                          Name Population
 2 ==================================================
 3      1024                Mumbai (Bombay)   10500000
 4      2331                          Seoul    9981619
 5       206                      São Paulo    9968485
 6      1890                       Shanghai    9696300
 7       939                        Jakarta    9604900
 8      2822                        Karachi    9269265
 9      3357                       Istanbul    8787958
10      2515               Ciudad de México    8591309
11      3580                         Moscow    8389200
12      3793                       New York    8008278
13      1532                          Tokyo    7980230
14      1891                         Peking    7472000
15       456                         London    7285000
16      1025                          Delhi    7206704
17       608                          Cairo    6789479
18      1380                        Teheran    6758845
19      2890                           Lima    6464693
20      1892                      Chongqing    6351600
21      3320                        Bangkok    6320174
22      2257              Santafé de Bogotá    6260862
```

First we need to import this SQL file into the SQLite database. Here's how to do it:

```
import sqlite3
import sys

if len(sys.argv) < 2:
print "Error: You must supply at least SQL script." print "Usage: %s table.db
./sql-dump.sql" % (sys.argv[0]) sys.exit(1)

script_path = sys.argv[1]

if len(sys.argv) == 3:
db = sys.argv[2]
else:
# if DB is not defined # create memory database db = ":memory:"

try: con = sqlite3.connect(db)
with con:

cur = con.cursor()
with open(script_path,'rb') as f: cur.executescript(f.read()) except sqlite3.Error as
err:
print "Error occured: %s" % err
```

This reads the SQL file and executes the SQL statements against the opened SQLite db file. If we don't specify the filename, SQLite creates the database in the memory. The statements are then executed line by line.

If we encounter any errors, we catch exceptions and print the error message to the user. After we have imported data into the database, we are able to query the data and do some processing. Here is the code to read the data from the database file:

```
import sqlite3
import sys

if len(sys.argv) != 2:
print "Please specify database file." sys.exit(1)

db = sys.argv[1]

try: con = sqlite3.connect(db)
with con:

cur = con.cursor()
query = 'SELECT ID, Name, Population FROM City ORDER BY

Population DESC LIMIT 1000'
con.text_factory = str cur.execute(query)

resultset = cur.fetchall()
# extract column names

col_names = [cn[0] for cn in cur.description] print "%10s %30s %10s" %
tuple(col_names) print "="*(10+1+30+1+10)

for row in resultset:
print "%10s %30s %10s" % row except sqlite3.Error as err:
print "[ERROR]:", err
```

How it works...

First, we verify that the user has provided the database filepath. This is just a quick check that we can proceed with the rest of the code.
Then, we try to connect to the database; if that fails, we catch sqlite3.Error and print it to the user.

If the connection is successful, we obtain a cursor using con.cursor(). A cursor is an iterator-like structure that enables us to traverse records of the result set returned from a database.

We define a query that we execute over the connection and we fetch the result set using cur.fetchall(). Had we expected just one result, we would have used just fetchone().

List comprehension over cur.description allows us to obtain column names. description is a read-only attribute and returns more than we need for just column names, so we just fetch the first item from every column's 7-item tuple.

We then use simple string formatting to print the header of our table with column names. After that, we iterate over resultset and print every row in a similar manner.

There's more...

Databases are the most common sources of data today. We could not present everything in this short recipe, but we can suggest where to look further.
The official Python documentation is the first place to look for an explanation about how to work with databases. The most common databases are open source databases—such as MySQL, PostgreSQL, and SQLite—and on the other end of the spectrum, there are enterprise database systems, such as MS SQL, Oracle, and Sybase. Mostly Python has support for them and the interface is abstracted always, so you don't have to change your program if your underlying database changes, but some tweaks may be required. It depends on whether you have used the specifics of a particular database system. For example, Oracle supports a specific language PL/SQL that is not standard SQL, and some things will not work if your database changes from Oracle to MS SQL. Similarly, SQLite does not support specifics from MySQL data types or database engine types (MyISAM and InnoDB). Those things can be annoying, but having your code rely on standard SQL (http://en.wikipedia.org/wiki/ SQL:2011) will make your code portable from one database system to another.

Cleaning up data from outliers

This recipe describes how to deal with datasets coming from the real world and how to clean them before doing any visualization.
We will present a few techniques, different in essence but with the same goal,

which is to get the data cleaned.

However, cleaning should not be fully automatic. We need to understand the data as given and be able to understand what the outliers are and what the data points represent before we apply any of the robust modern algorithms made to clean the data. This is not something that can be defined in a recipe because it relies on vast areas such as statistics, knowledge of the domain, and a good eye (and then some luck).

Getting ready

We will use the standard Python modules we already know about, so no additional installation is required.

In this recipe, I will introduce a new term, MAD. Median absolute deviation (MAD) in statistics represents a measure of the variability of a univariate (possessing one variable) sample of quantitative data. It is a measure of statistical dispersion. It falls into a group of robust statistics, such that it is more resilient to outliers.

How to do it...

Here's one example that shows how to use MAD to detect outliers in our data. We will perform the following steps for this:

1. Generate normally distributed random data.
2. Add in a few outliers.
3. Use the function is_outlier() to detect the outliers.
4. Plot both the datasets (x and filtered) to see the difference.

```
import numpy as np
import matplotlib.pyplot as plt

def is_outlier(points, threshold=3.5):
"""
Returns a boolean array with True if points are outliers and

False
otherwise.

Data points with a modified z-score greater than this # value will be classified as outliers.
```

```
"""
# transform into vector
if len(points.shape) == 1:

points = points[:,None]
# compute median value
median = np.median(points, axis=0)

# compute diff sums along the axis
diff = np.sum((points - median)**2, axis=-1) diff = np.sqrt(diff)
# compute MAD
med_abs_deviation = np.median(diff)

# compute modified Z-score

# http://www.itl.nist.gov/div898/handbook/eda/section4/eda43. htm#Iglewicz
modified_z_score = 0.6745 * diff / med_abs_deviation

# return a mask for each outlier return modified_z_score > threshold
# Random data
x = np.random.random(100)
# histogram buckets buckets = 50
# Add in a few outliers
x = np.r_[x, -49, 95, 100, -100]

# Keep valid data points
# Note here that
# "~" is logical NOT on boolean numpy arrays filtered = x[~is_outlier(x)]
# plot histograms plt.figure()

plt.subplot(211)
plt.hist(x, buckets) plt.xlabel('Raw')

plt.subplot(212)
plt.hist(filtered, buckets) plt.xlabel('Cleaned')

plt.show()
```

Note that, in NumPy, the ~ operator is overloaded to operate as a logical operator, not on Boolean arrays. For example, suppose we fire up IPython in pylab mode.

$ ipython –pylab
We get:

In [1]: ~numpy.array(False)
Out[1]: True

We should see two distinct histograms, the first one showing almost nothing—except in one bucket where the biggest outlier is—and the second histogram showing diversified data buckets because we removed outliers.

Another way to identify outliers is to visually inspect your data. In order to do so, we could create scatter plots, where we could easily spot values that are out of the central swarm. We can also make a box plot, which will display the median, quartiles above and below the median, and points that are distant to this box.

The box extends from the lower to upper quartile values of the data, with a line

at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

Here's an example to demonstrate that:

```
from pylab import *

# fake up some data spread= rand(50) * 100
center = ones(25) * 50

# generate some outliers high and low flier_high = rand(10) * 100 + 100
flier_low = rand(10) * -100

# merge generated data set
data = concatenate((spread, center, flier_high, flier_low), 0)

subplot(311)
# basic plot
# 'gx' defining the outlier plotting properties boxplot(data, 0, 'gx')

# compare this with similar scatter plot
subplot(312)
spread_1 = concatenate((spread, flier_high, flier_low), 0) center_1 = ones(70) *
25
scatter(center_1, spread_1)
xlim([0, 50])

# and with another that is more appropriate for # scatter plot
subplot(313)
center_2 = rand(70) * 50
scatter(center_2, spread_1)
xlim([0, 50])

show()
```

We can then see x-shaped markers representing outliers:

We can also see that the second plot showing a similar dataset in the scatter plot is not very intuitive because the x axis has all the values at 25 and we don't really distinguish between inliers and outliers.

The third plot, where we generated values on the x axis to be spread across the range from 0 to 50, gives us more visibility of the different values and we can see what values are outliers in terms of the y axis.

In the following code example, we see how the same data (in this example, uniformly distributed) can display itself very differently and sometimes deceptively convey some information that is not really true:

```
# generate uniform data points
x = 1e6*rand(1000)
y = rand(1000)

figure()
```

```
# crate first subplot subplot(211)
# make scatter plot scatter(x, y)
# limit x axis
xlim(1e-6, 1e6)

# crate second subplot
subplot(212)
# make scatter plot
scatter(x,y)
# but make x axis logarithmic xscale('log')
# set same x axis limit
xlim(1e-6, 1e6)

show()
```

This is the resulting output:

What if we have a dataset with missing values? We can use NumPy loaders to compensate for missing values, or we can write code to replace existing values

with the ones we need for further use.

Say we want to illustrate some dataset over the geographical map of USA and may have values for state names that are not consistent in the dataset. For example, we have values OH, Ohio, OHIO, US-OH, and OH-USA all representing the state of Ohio in the USA. What we must do in this situation is inspect the dataset manually, either loading it in a spreadsheet processor such as Microsoft Excel or OpenOffice.org Calc. Sometimes, it is easy enough to just print all the lines using Python. If the file is CSV or CSV-like, we can open it with any text editor and inspect the data directly.

After we have concluded about what is in the data, we can write Python code to group those similar values and replace them with the one value that is going to make further processing consistent. The usual way of doing this is to read in lines of the file using readlines() and use standard Python string manipulation functions to perform manipulations.

There's more...

There are special products, both commercial and non-commercial (such as OpenRefine—https://github.com/OpenRefine), that provide some automation around transformation on "dirty" live datasets.

Still, manual work is involved, depending on how noisy the data is and how great our understanding of that data is.
If you want to find out more about cleaning outliers and cleaning of data in general, look for statistical models and the sampling theory.

Reading files in chunks

Python is very good at handling reading and writing files or file-like objects. For example, if you try to load big files, say a few hundred MB, assuming you have a modern machine with at least 2 GB of RAM, Python will be able to handle it without any issue. It will not try to load everything at once, but play smart and load it as needed.

So even with decent file sizes, doing something as simple as the following code will work straight out of the box:

```
with open('/tmp/my_big_file', 'r') as bigfile:
```

```
for line in bigfile:
# line based operation, like 'print line'
```

But if we want to jump to a particular place in the file or do other nonsequential reading, we will need to use the handcrafted approach and use IO functions such as seek(), tell(), read(), and next() that allow enough flexibility for most users. Most of these functions are just bindings to C implementations (and are OS-specific), so they are fast, but their behavior can vary based on the OS we are running.

How to do it...

Depending on what our aim is, processing large files can sometimes be managed in chunks. For example, you could read 1,000 lines and process them using Python standard iterator-based approaches.

```
import sys
filename = sys.argv[1] # must pass valid file name

with open(filename, 'rb') as hugefile:
chunksize = 1000
readable = ''
# if you want to stop after certain number of blocks # put condition in the while
while hugefile:

# if you want to start not from 1st byte
# do a hugefile.seek(skipbytes) to skip
# skipbytes of bytes from the file start
start = hugefile.tell()
print "starting at:", start
file_block = '' # holds chunk_size of lines for _ in xrange(start, start +
chunksize):

line = hugefile.next()
file_block = file_block + line
print 'file_block', type(file_block), file_block

readable = readable + file_block
# tell where are we in file
# file IO is usually buffered so tell()
```

```
# will not be precise for every read.
stop = hugefile.tell()
print 'readable', type(readable), readable print 'reading bytes from %s to %s' %
(start, stop) print 'read bytes total:', len(readable)

# if you want to pause read between chucks # uncomment following line
#raw_input()
```

We call this code from the Python command-line interpreter, giving the filename path as the first parameter.

$ python ch02-chunk-read.py myhugefile.dat

How it works...

We want to be able to read blocks of lines for processing without reading the whole file in the memory.

We open the file and read in lines in the inner for loop. The way we move through the file is by calling next() on the file object. This function reads the line from the file and moves the file pointer to the next line. We append lines in the file_block variable during the loop execution. In order to simplify the example code, we don't do any processing but just add file_block to complete the output variable readable.

We do some printing during execution just to illustrate the current state of certain variables. The last comment line in the while loop, raw_input(), can be uncommented and we can pause the execution and read the printed lines above it.

There's more...

This recipe is, of course, just one of the possible approaches to reading large (huge) files. Other approaches could include specific Python or C libraries, but they all depend on what it is that we aim to do with data and how we want to process it.

Parallel approaches such as the MapReduce paradigm have become very popular recently as we get more processing power and memory for a low price.

Multiprocessing is also a feasible approach sometimes, as Python has good library support for creating and managing threads with several libraries such as multiprocessing, threading, and thread.

If processing huge files is a repeated process for a project, we suggest building your data pipeline so that every time you need data ready in a specific format on the output end, you don't have to go to the source and do it manually.

Reading streaming data sources

What if the data that is coming from the source is continuous? What if we need to read continuous data? This recipe will demonstrate a simple solution that will work for many common real-life scenarios, though it is not universal and you will need to modify it if you hit a special case in your application.

How to do it...

In this recipe, we will show you how to read an always-changing file and print the output. We will use the common Python module to accomplish this.

```
import time
import os
import sys

if len(sys.argv) != 2:
print >> sys.stderr, "Please specify filename to read"
filename = sys.argv[1]
if not os.path.isfile(filename):
print >> sys.stderr, "Given file: \"%s\" is not a file" % filename

with open(filename,'r') as f:
# Move to the end of file
filesize = os.stat(filename)[6]
f.seek(filesize)

# endlessly loop
while True:
where = f.tell()
# try reading a line
line = f.readline()
# if empty, go back
if not line:
time.sleep(1)
f.seek(where)
```

```
else:
# , at the end prevents print to add newline, as readline()
# already read that. print line,
```

How it works...

The core of the code is inside the while True: loop. This loop never stops (unless we interrupt it by pressing *Ctrl + C* on our keyboard). We first move to the end of the file we are reading and then we try to read a line. If there is no line, that means nothing was added to the file after we checked it using seek(). So, we sleep for one second and then try again.

If there is a nonempty line, we print that out and suppress the new line character.

There's more...

We might want to read the last *n* lines. We could do that by going almost to the end of the file. We could go there by looking for the file, that is, with file.seek(filesize – N * avg_ line_len). Here, avg_line_len should be the approximation of average line length in that file (approximately 1,024). Then, we could use readlines() from that point to read line, and then print just [-N] lines from that list.

The idea from this example can be used for various solutions. For example, the input has to be a file-like object or a remote HTTP-accessible resource. Thus, one can read the input from a remote service and continuously parse it and update live charts, for example, or update the intermediate queue, buffer, or database.

One particular module is very useful for stream handling—io. It is in Python from Version 2.6, is built as a replacement for the file module, and is a default interface in Python 3.x.

In some more complex data pipelines, we will need to enable some sort of message queues, where our incoming continuous data will have to be queued for some time before we are able to accept it. This enables us, as consumers of the data, to be able to pause processing if we are overloaded. Having data on the common message bus enables other clients on the project to consume the same data and not interfere with our software.

Importing image data into NumPy arrays

We are going to demonstrate how to do image processing using Python's libraries such as NumPy and SciPy.

In scientific computing, images are usually seen as *n*-dimensional arrays. They are usually two-dimensional arrays; in our examples, they are represented as a NumPy array data structure. Therefore, functions and operations performed on those structures are seen as matrix operations.

Images in this sense are not always two-dimensional. For medical or bio-sciences, images are data structures of higher dimensions, such as 3D (having the z axis as depth or as the time axis) or 4D (having three spatial dimensions and a temporal one as the fourth dimension). We will not be using those in this recipe.

We can import images using various techniques; they all depend on what you want to do with image. Also, it depends on the larger ecosystem of tools you are using and the platform you are running your project on.

In this recipe we will demonstrate several ways to use image processing in Python, mainly related to scientific processing and less on the artistic side of image manipulation.

Getting ready

In some examples in this recipe, we use the SciPy library, which you have already installed if you have installed NumPy. If you haven't, it is easily installable using your OS's package manager by executing the following command:

$ sudo apt-get install python-scipy

For Windows users, we recommend using prepackaged Python environments, such as EPD, that we discussed in *Chapter 1*, *Preparing Your Working Environment*.

If you want to install these using official source distributions, make sure you have installed system dependencies, such as:

f BLAS and LAPACK: libblas and liblapack

f C and Fortran compilers: gcc and gfortran

How to do it...

Whoever has worked in the field of digital signal processing or even attended a university course on this or a related subject must have come across Lena's image, the de facto standard image, used for demonstrating image processing algorithms.

SciPy contains this image already packed inside the misc. module, so it is really simple for us to re-use that image. This is how to read and show this image:

```
import scipy.misc
import matplotlib.pyplot as plt
# load already prepared ndarray from scipy lena = scipy.misc.lena()
# set the default colormap to gray plt.gray()

plt.imshow(lena) plt.colorbar() plt.show()
```

This should open a new window with a figure displaying Lena's image in gray tones and axes. The color bar shows a range of values in the figure; here it shows 0—black to 255—white.

Further, we could examine this object with the following code:

```
print lena.shape
print lena.max()
print lena.dtype
```

The output for the preceding code is as follows:

```
(512, 512)
245
dtype('int32')
```

Here we see that the image is:

f 512 points wide and 512 points high
f The max value in the whole array (that is, the image) is 254
f Every point is represented as a little endian 32-bit long integer

We could also read in an image using Python Imaging Library (PIL), which we installed in *Chapter 1*, *Preparing Your Working Environment*.

```
import numpy
import Image
import matplotlib.pyplot as plt

bug = Image.open('stinkbug.png')
arr = numpy.array(bug.getdata(), numpy.uint8).reshape(bug.size[1], bug.size[0],
3)

plt.gray()
plt.imshow(arr) plt.colorbar() plt.show()
```

We should see something similar to Lena's image as follows:

This is useful if we are already tapping into an existing system that uses PIL as their default image loader.

How it works...

Other than just loading the images, what we really want to do is use Python to manipulate images and process them. We want to be able to, say, load a real image that consists of RGB channels, convert that into one channel ndarray, and later use array slicing to also zoom in to the part of the image. Here's the code to demonstrate how we are able to use NumPy and matplotlib to do that.

```
import matplotlib.pyplot as plt
import scipy
import numpy

bug = scipy.misc.imread('stinkbug1.png')

# if you want to inspect the shape of the loaded image # uncomment following line
```

#print bug.shape

the original image is RGB having values for all three # channels separately. We need to convert that to greyscale image # by picking up just one channel.

convert to gray bug = bug[:,:,0] bug[:,:,0]
is called array slicing. This NumPy feature allows us to select any part of the multidimensional array. For example, let's see a one-dimensional array:

```
>>> a = array(5, 1, 2, 3, 4)
>>> a[2:3]
array([2])
>>> a[:2]
array([5, 1])
>>> a[3:]
array([3, 4])
```

For multidimensional arrays, we separate each dimension with a comma (,). For example:

```
>>> b = array([[1,1,1],[2,2,2],[3,3,3]]) # matrix 3 x 3
>>> b[0,:] # pick first row
array([1,1,1])
>>> b[:,0] # we pick the first column
array([1,2,3])
```

Have a look at the following code:

# show original image plt.figure()
plt.gray()

plt.subplot(121) plt.imshow(bug)
# show 'zoomed' region zbug = bug[100:350,140:350]

Here we zoom into the particular portion of the whole image. Remember that the image is just a multidimensional array represented as a NumPy array. Zooming here means selecting a range of rows and columns from this matrix. So we select a partial matrix from rows 100 to 250 and columns 140 to 350. Remember that indexing starts at 0, so the row at coordinate 100 is the 101st row.

```
plt.subplot(122)
plt.imshow(zbug)
plt.show()
```
This will be displayed as follows:

There's more...

For large images we recommend using numpy.memmap for memory mapping of images. This will speed up manipulating the image data. For example:

```
import numpy
file_name = 'stinkbug.png'
image = numpy.memmap(file_name, dtype=numpy.uint8, shape = (375, 500))
```

Here we load part of a large file into memory, accessing it as a NumPy array. This is very efficient and allows us to manipulate file data structures as standard NumPy arrays without loading everything into memory. The argument shape defines the shape of the array loaded from the file_name argument, which is a file-like object. Note that this is a similar concept to Python's mmap argument (http://docs.python.org/2/library/mmap.html) but is different in a very important way—NumPy's memmap attribute returns an array-like object, while Python's mmap returns a file-like object. So the way we use them is very different, yet very natural in each environment.

There are some specialized packages that just focus on image processing, for

example, scikit-image (http://scikit-image.org/); this is basically a free collection of algorithms for image processing, built on top of NumPy/SciPy libraries. If you want to do edge detection, remove noise from an image, or find contours, scikit is the tool to use to look for algorithms. The best way to start is to look at the example gallery and find the example image and code (http://scikit-image.org/docs/dev/auto_examples/).

Generating controlled random datasets

In this recipe, we will show different ways of generating random number sequences and word sequences. Some of the examples use standard Python modules, and some use NumPy/SciPy functions.

We will go into some statistics terminology, but we will explain every term so you don't have to have a statistical reference book with you while reading this recipe.

We generate artificial datasets using common Python modules. By doing so, we are able to understand distributions, variance, sampling, and similar statistical terminology. More importantly, we can use this fake data as a way to understand if our statistical method is capable of discovering models we want to discover. We can do that because we know the model in advance and verify our statistical method by applying it over our known data. In real life, we don't have that ability and there is always a percentage of uncertainty that we must assume, giving way to errors.

Getting ready

We don't need anything new installed on the system in order to exercise these examples. Having some knowledge of statistics is useful, although not required. To refresh our statistical knowledge, here's a little glossary we will use in this and the following chapters.

f Distribution or probability distribution: This links the outcome of a statistical experiment with the probability of occurrence of that experiment.

f Standard deviation: This is a numerical value that indicates how individuals vary in comparison to a group. If they vary more, the standard derivation will be big, and the opposite—if all the individual experiments are more or less the same across the whole group, the standard derivation will be small.

f Variance: This equals the square of standard derivation.

f Population or statistical population: This is a total set of all the potentially observable cases, for example, all the grades of all the students in the world, if we are interested in getting the student average of the world.

f Sample: This is a subset of the population. We cannot obtain all the grades of all the students in the world, so we have to gather only a sample of data and model it.

How to do it...

We can generate a simple random sample using Python's modulerandom. Here's an example:

```
import pylab
import random
SAMPLE_SIZE = 100

# seed random generator
# if no argument provided
# uses system current time
random.seed()

# store generated random values here
real_rand_vars = []

# pick some random values
real_rand_vars = [random.random() for val in xrange(SIZE)] # create histogram from data in 10 buckets
pylab.hist(real_rand_vars, 10)

# define x and y labels pylab.xlabel("Number range") pylab.ylabel("Count")

# show figure pylab.show()
```

This is a uniformly distributed sample. When we run this example, we should see something similar to the following plot:

Try setting SAMPLE_SIZE to a big number (say 10000) and see how the histogram behaves.

If we wanted to have values that range not from 0 to 1, but say, from 1 to 6 (simulating single dice throws, for example), we could use random.randint(min, max); here, min and max are the lower and upper inclusive bounds respectively. If what you want to generate are floats and not integers, there is a random.uniform(min, max) function to provide that. In a similar fashion, and using the same tools, we can generate a time series plot of fictional price growth data with some random noise.

```
import pylab
import random

# days to generate data for duration = 100
# mean value
mean_inc = 0.2

# standard deviation std_dev_inc = 1.2
```

```
# time series
x = range(duration) y = []
price_today = 0

for i in x:
next_delta = random.normalvariate(mean_inc, std_dev_inc) price_today +=
next_delta
y.append(price_today)

pylab.plot(x,y)
pylab.xlabel("Time") pylab.xlabel("Time") pylab.ylabel("Value") pylab.show()
```

This code defines a series of 100 data points (fictional days). For every next day, we pick a random value from the normal distribution (random.normalvariate()) ranging from mean_inc tostd_dev_inc and add that value to yesterday's price value (price_today).

If we wanted more control, we could use different distributions. The following code illustrates and visualizes different distributions. We will comment separate code sections as we present them. We start by importing required modules and defining a number of histogram buckets. We also create a figure that will hold our histograms.

```
# coding: utf-8
import random
import matplotlib
import matplotlib.pyplot as plt

SAMPLE_SIZE = 1000 # histogram buckets buckets = 100

plt.figure()
# we need to update font size just for this example
matplotlib.rcParams.update({'font.size': 7})
```

To lay out all the required plots, we define a grid of six by two subplots for all the histograms. The first plot is a normal distributed random variable.

```
plt.subplot(621)
plt.xlabel("random.random")
# Return the next random floating point number in the range [0.0, 1.0).
res = [random.random() for _ in xrange(1, SAMPLE_SIZE)]
```

plt.hi

For the second plot, we plot a uniformly distributed random variable.

```
plt.subplot(622)
plt.xlabel("random.uniform")
# Return a random floating point number N such that a <= N <= b for a <= b and
b <= N <= a for b < a.
# The end-point value b may or may not be included in the range depending on
floating-point rounding in the equation a + (b-a) * random().
a = 1
b = SAMPLE_SIZE
res = [random.uniform(a, b) for _ in xrange(1, SAMPLE_SIZE)] plt.hist(res,
buckets)
```

The third plot is a triangular distribution.

```
plt.subplot(623)
plt.xlabel("random.triangular")

# Return a random floating point number N such that low <= N <= high and with
the specified # mode between those bounds. The low and high bounds default to
zero and one. The mode
# argument defaults to the midpoint between the bounds, giving a symmetric
distribution.
low = 1
high = SAMPLE_SIZE
res = [random.triangular(low, high) for _ in xrange(1, SAMPLE_SIZE)]
plt.hist(res, buckets)
```

The fourth plot is a beta distribution. The condition on the parameters is that alpha and beta should be greater than zero. The returned values range between 0 and 1.

```
plt.subplot(624)
plt.xlabel("random.betavariate")
alpha = 1
beta = 10
res = [random.betavariate(alpha, beta) for _ in xrange(1, SAMPLE_ SIZE)]
plt.hist(res, buckets)
```

The fifth plot visualizes an exponential distribution. lambd is 1.0 divided by the desired mean. It should be nonzero. (The parameter would be called lambda, but that is a reserved word in Python.) The returned values range from 0 to positive infinity if lambd is positive, and from negative infinity to 0 if lambd is negative.

```
plt.subplot(625)
plt.xlabel("random.expovariate")
lambd = 1.0 / ((SAMPLE_SIZE + 1) / 2.)
res = [random.expovariate(lambd) for _ in xrange(1, SAMPLE_SIZE)]
plt.hist(res, buckets)
```

Our next plot is the gamma distribution, where the condition on the parameters is that alpha and beta are greater than 0. The probability distribution function is:

$-x$

x

$a-1_e \beta$

$PDF\ x(\ )=\gamma(a)a$

β

Here's the code for the gamma distribution:

```
plt.subplot(626)
plt.xlabel("random.gammavariate")

alpha = 1
beta = 10
res = [random.gammavariate(alpha, beta) for _ in xrange(1, SAMPLE_ SIZE)]
plt.hist(res, buckets)
```

Log normal distribution is our next plot. If you take the natural logarithm of this distribution, you'll get a normal distribution with the mean mu and the standard deviation sigma. mu can have any value, and sigma must be greater than zero.

```
plt.subplot(627)
plt.xlabel("random.lognormvariate")
mu = 1
sigma = 0.5
res = [random.lognormvariate(mu, sigma) for _ in xrange(1, SAMPLE_ SIZE)]
plt.hist(res, buckets)
```

The next plot is normal distribution, where mu is the mean and sigma is the standard deviation.

```
plt.subplot(628)
plt.xlabel("random.normalvariate")
mu = 1
sigma = 0.5
res = [random.normalvariate(mu, sigma) for _ in xrange(1, SAMPLE_ SIZE)]
plt.hist(res, buckets)
```

The last plot is the Pareto distribution. alpha is the shape parameter.

```
plt.subplot(629)
plt.xlabel("random.paretovariate")
alpha = 1
res = [random.paretovariate(alpha) for _ in xrange(1, SAMPLE_SIZE)]
plt.hist(res, buckets)

plt.tight_layout() plt.show()
```

This was a big code example, but basically, we pick 1,000 random numbers according to various distributions. These are common distributions used in different statistical branches (economics, sociology, bio-sciences, and so on). We should see differences in the histogram based on the distribution algorithm used. Take a moment to understand the following nine plots:

Use seed() to initialize the pseudo-random generator, so random() produces the same expected random values. This is sometimes useful and it is better than pregenerating random data and saving it to a file. The latter technique is not always feasible as it requires saving (possibly huge amounts of) data on a filesystem.

If you want to prevent any repeatability of your randomly generated sequences, we recommend using random.SystemRandom, which uses os.urandom underneath; os.urandom provides access to more entropy sources. If using this random generator interface, seed() and setstate() have no effect; hence, these samples are not reproducible.

If we want to have some random words, the easiest way (on Linux) is probably to use /usr/share/dicts/words. We can see how that is done, in the following example:

```
import random

with open('/usr/share/dict/words', 'rt') as f:
words = f.readlines()
words = [w.rstrip() for w in words]

for w in random.sample(words, 5):
print w
```

This solution is for Unix only and will not work on Windows (it will work on Mac OS, though). For Windows, you could use a file constructed from various free sources (Project Gutenberg, Wiktionary, British National Corpus, or http://norvig.com/big.txt by Dr Peter Norvig).

Smoothing the noise in real-world data

In this recipe, we introduce a few advanced algorithms to help with cleaning the data coming from real-world sources. These algorithms are well known in the signal processing world, and we will not go deep into mathematics but will just exemplify how and why they work and for what purposes they can be used.

Getting ready

Data that comes from different real-life sensors usually is not smooth and clean

and contains some noise that we usually don't want to show on diagrams and plots. We want graphs and plots to be clear and to display information and cost viewers minimal efforts to interpret.

We don't need any new software installed because we are going to use some already familiar Python packages: NumPy, SciPy, and matplotlib.

How to do it...

The basic algorithm is based on using the rolling window (for example, convolution). This window rolls over the data and is used to compute the average over that window.

For our discrete data, we use NumPy's convolve function; it returns a discrete linear convolution of two one-dimensional sequences. We also use NumPy's linspace function, which generates a sequence of evenly spaced numbers for a specified interval.

The function ones defines an array or matrix (for example, a multidimensional array) where every element has the value 1. This helps with generating windows for use in averaging.

How it works...

One simple and naive technique to smooth the noise in data we are processing is to average over some window (sample) and plot just that average value for the given window, instead of all the data points. This is the basis for more advanced algorithms.

```
from pylab import *
from numpy import *

def moving_average(interval, window_size): '''Compute convoluted window for given size '''
window = ones(int(window_size)) / float(window_size) return convolve(interval, window, 'same')

t = linspace(-4, 4, 100) y = sin(t) + randn(len(t))*0.1
plot(t, y, "k.")

# compute moving average y_av = moving_average(y, 10) plot(t, y_av,"r")
#xlim(0,1000)
```

```
xlabel("Time") ylabel("Value") grid(True)
show()
```

Here, we show how the smoothed line looks compared to the original data points (plotted as dots):

Following on this idea, we can jump ahead to an even more advanced example and use the existing SciPy library to make this window smoothing work even better.

The method we are going to demonstrate is based on convolution (summation of functions) of a scaled window with the signal (that is, data points). This signal is prepared in a clever way, adding copies of the same signal on both ends but reflecting it, so we minimize the boundary effect. This code is based on SciPy Cookbook's example that can be found here: http://www.scipy.org/Cookbook/SignalSmooth.

```
import numpy
```

```
from numpy import *
from pylab import *

# possible window type
WINDOWS = ['flat', 'hanning', 'hamming', 'bartlett', 'blackman'] # if you want to
see just two window type, comment previous line, # and uncomment the
following one
# WINDOWS = ['flat', 'hanning']

def smooth(x, window_len=11, window='hanning'): """
Smooth the data using a window with requested size. Returns smoothed signal.

x -- input signal
window_len -- lenght of smoothing window
window -- type of window: 'flat', 'hanning', 'hamming',

'bartlett', 'blackman'
flat window will produce a moving average smoothing. """
if x.ndim != 1:
raise ValueError, "smooth only accepts 1 dimension arrays."
if x.size < window_len:
raise ValueError, "Input vector needs to be bigger than window size."
if window_len < 3: return x
if not window in WINDOWS:
raise ValueError("Window is one of 'flat', 'hanning',

'hamming', "
'"bartlett', 'blackman'") # adding reflected windows in front and at the end
s=numpy.r_[x[window_len-1:0:-1], x, x[-1:-window_len:-1]] # pick windows
type and do averaging
if window == 'flat': #moving average

w = numpy.ones(window_len, 'd')
else:
# call appropriate function in numpy
w = eval('numpy.' + window + '(window_len)')

# NOTE: length(output) != length(input), to correct this: # return
y[(window_len/2-1):-(window_len/2)] instead of just y. y =
numpy.convolve(w/w.sum(), s, mode='valid')
```

```
return y

# Get some evenly spaced numbers over a specified interval. t = linspace(-4, 4,
100)

# Make some noisy sinusoidal x = sin(t)
xn = x + randn(len(t))*0.1

# Smooth it y = smooth(x)
# windows ws = 31
subplot(211) plot(ones(ws))
# draw on the same axes hold(True)
# plot for every windows for w in WINDOWS[1:]:
eval('plot('+w+'(ws) )') # configure axis properties axis([0, 30, 0, 1.1])
# add legend for every window legend(WINDOWS)
title("Smoothing windows")

# add second plot subplot(212)
# draw original signal plot(x)

# and signal with added noise plot(xn)
# smooth signal with noise for every possible windowing algorithm for w in
WINDOWS:
plot(smooth(xn, 10, w))

# add legend for every graph
l=['original signal', 'signal with noise'] l.extend(WINDOWS)
legend(l)

title("Smoothed signal")
show()
```

We should see the following two plots to see how the windowing algorithm influences the noise signal. The top plot represents possible windowing algorithms and the bottom one displays every possible result, from the original signal, to the noised up signal, and even the smoothed signal for every windowing algorithm. Try commenting possible window types and leave just one or two to gain better understanding.

There's more...

Another very popular signal smoothing algorithm is Median Filter. The main idea of this filter is to run through the signal entry by entry, replacing each entry with the median of neighboring entries. This idea makes this filter fast and usable both for one-dimensional datasets as well as for two-dimensional datasets (such as images).

In the following example, we use the implementation from the SciPy signal toolbox:

```
import numpy as np
import pylab as p
import scipy.signal as signal

# get some linear data
x = np.linspace (0, 1, 101)
# add some noisy signal
x[3::10] = 1.5

p.plot(x)
p.plot(signal.medfilt(x,3))
```

```
p.plot(signal.medfilt(x,5))

p.legend(['original signal', 'length 3','length 5'])
p.show ()
```

We see in the following plot that the bigger the window, the more our signal gets distorted as compared to the original but the smoother it looks:

There are many more ways to smooth data (signals) that you receive from external sources. It depends a lot on the area you are working in and the nature of the signal. Many algorithms are specialized for a particular signal, and there may not be a general solution for every case you encounter.

There is, however, one important question: "When should you not smooth a signal?" One common situation where you should not smooth signals is prior to statistical procedures, such as least-squares curve fitting, because all smoothing algorithms are at least slightly lossy and they change the signal shape. Also, smoothed noise may be mistaken for an actual signal.

3

Drawing Your First Plots and Customizing Them

In this chapter we will go into a lot more detail and present most of the possibilities of matplotlib. We will cover:

f Defining plot types – bar, line, and stacked charts
f Drawing simple sine and cosine plots
f Defining axis lengths and limits
f Defining plot line styles, properties, and format strings
f Setting ticks, labels, and grids
f Adding legends and annotations
f Moving spines to the center
f Making histograms
f Making bar charts with error bars
f Making pie charts count
f Plotting with filled areas
f Drawing scatter plots with colored markers

Introduction

Although we have already drawn our first plots using matplotlib, we didn't go into the details about how they work, how to set them up, or what the possibilities are with using matplotlib. We explore and exercise most common types of data visualizations: line graphs, bar charts, histograms, pies, and variations thereof.

matplotlib is a powerful toolbox that satisfies almost all our needs for 2D and some 3D plotting needs as well. The best way the authors intend for you to learn matplotlib is through examples. When we need to draw a plot, we look for a similar example and try to change it to fit our needs. In this way, we are also going to present you with some useful examples and believe that this example will help you find a plot most similar to what you need.

Defining plot types – bar, line, and stacked charts

In this recipe, we will present different basic plots and what are they used for. Most of the plots described here are used daily, and some of them present the basis for understanding more advanced concepts in data visualization.

Getting ready

We start with some common charts from the matplotlib.pyplot library with just sample datasets to get started with basic charting and lay down the foundations of the following recipes.

How to do it...

We start by creating a simple plot in IPython. IPython is great because it allows us to interactively change plots and see the results immediately.

1. Start IPython by typing the following at the command prompt: **$ ipython --pylab**

2. Then type the matplotlib plot code:

```
In [1]: plot([1,2,3,2,3,2,2,1])
Out[1]: [<matplotlib.lines.Line2D at 0x412fb50>]
```

The plot should open in a new window displaying the default look of the plot and some supporting information:

The basic plot in matplotlib contains the following elements:
f x and y axes: These are both horizontal and vertical axes.
f x and y tickers: These are little tickers denoting the segments of axes. There can be major and minor tickers.
f x and y tick labels: These represent values on particular axis.
f Plotting area: This is where the actual plots are drawn.

You will notice that the values we provided to plot() are y axis values. plot()provides default values for the x axis; they are linear values from 0 to 7 (the number of y values minus 1).

Now, try adding values for the x axis; as first argument to the plot() function, again in the same IPython session, type:
In [2]: plot([4,3,2,1],[1,2,3,4])
Out[2]: [<matplotlib.lines.Line2D at 0x31444d0>]

Note how IPython counts input and output lines (In [2] and Out [2]). This will help us remember where we are in the current session, and enables more advanced features such as saving part of the session in a Python file. During data

analysis, using IPython for prototyping is the fastest way to come to a satisfying solution and then save particular sessions into a file, to be executed later if you need to reproduce the same plot.

This will update the plot to look like this:

We see here how matplotlib expands the y axis to accommodate the new value range and automatically changes color of the second plot line to enable us to distinguish the new plot.

Unless we turn off the hold property (by calling hold(False)) all subsequent plots will draw over the same axes. This is the default behavior in pylab mode in IPython, while in regular Python scripts, hold is off by default.

Let us pack some more common plots and compare them over the same dataset. You can type this in IPython or run it from a separate Python script:

```
from matplotlib.pyplot import *

# some simple data
x = [1,2,3,4]
y = [5,4,3,2]
# create new figure figure()

# divide subplots into 2 x 3 grid # and select #1
```

```
subplot(231)
plot(x, y)

# select #2 subplot(232) bar(x, y)

# horizontal bar-charts subplot(233)
barh(x, y)

# create stacked bar charts subplot(234)
bar(x, y)

# we need more data for stacked bar charts y1 = [7,8,5,3]
bar(x, y1, bottom=y, color = 'r')

# box plot subplot(235) boxplot(x)

# scatter plot subplot(236) scatter(x,y)

show()
```

This is how it should turn out into graphs:

How it works...

With figure() we create a new figure. If we supply a string argument, such as sample charts, it will be the backend title of a window. If we call the figure() function with the same parameter (that can also be a number), we will make the corresponding figure active and all the following plotting will be performed on that figure.

Next, we divide the figure into a 2 by 3 grid using a subplot(231) call. We could call this using subplot(3, 2, 1), where the first parameter is the number of rows, the second is the number of columns, and the third represents the plot number.

We continue and create a common charting type using simple calls to create vertical bar charts (bar()) and horizontal bars (barh()). For stacked bar charts, we need to tie two bar chart calls together. We do that by connecting the second bar chart with the previous using the parameter bottom = y.
Box plots are created using the boxplot() call, where the box extends from lower

to upper quartiles with the line at the median value. We will return to box plots shortly.

We finally create a scatter plot to give you an idea of a point-based dataset. This is probably more appropriately used when we have thousands of data points in a dataset, but here we wanted to illustrate the difference in representations of the same dataset.

There's more...

We can return to box plots now as we need to explain the most important display options.

For starters, we could add whiskers that extend from the box to represent the whole range of the dataset. Box and whiskers plots are mainly used to represent variations of data in one or multiple datasets; they are easy to compare and easy to read. In the same plot they can represent five statistics:

f Minimum value: This is in the dataset
f Second quartile: Below this the lower 25 percent of the given dataset lies f Median value: This is the median value of the dataset
f Third quartile: Above this the upper 25 percent of the given dataset lies f Maximum value: This is the maximum value of the given dataset

To illustrate this behavior, we will demonstrate plotting the same dataset in a box plot and a histogram as in the following code:

```
from pylab import *

dataset = [113, 115, 119, 121, 124,
124, 125, 126, 126, 126,
127, 127, 128, 129, 130,
130, 131, 132, 133, 136]

subplot(121)
boxplot(dataset, vert=False)
subplot(122) hist(dataset)
show()
```

That will give us the following plots:

In the preceding comparison, we can observe a difference in representation of the same dataset in two different charts. The one on the left points toward the five mentioned statistical values, while the one on the right (the histogram) displays the grouping of the dataset in a given range.

Drawing a simple sine and cosine plot

This recipe will go over basics of plotting mathematical functions and several things that are related to math graphs, such as writing Greek symbols in labels and on curves.

Getting ready

The most common graph we will use is the line plot command, which draws the given (x,y) coordinates on a figure plot.

How to do it...

We start with computing sine and cosine functions over the same linear interval —from Pi to Pi, with 256 points in between—and we plot the values for sin(x) and cos(x) over the same plot: import matplotlib.pyplot as pl

```
import numpy as np
x = np.linspace(-np.pi, np.pi, 256, endpoint=True)
y = np.cos(x) y1 = np.sin(x)
pl.plot(x,y) pl.plot(x, y1)
```

pl.show() That will give us the following graph:

Following this simple plot, we can customize more to give more information and be more precise about axes and boundaries:

```
from pylab import *
import numpy as np

# generate uniformly distributed
# 256 points from -pi to pi, inclusive
x = np.linspace(-np.pi, np.pi, 256, endpoint=True)

# these are vectorised versions
# of math.cos, and math.sin in built-in Python maths # compute cos for every x
y = np.cos(x)

# compute sin for every x y1 = np.sin(x)
# plot cos plot(x, y)
# plot sin plot(x, y1)
```

```
# define plot title
title("Functions $\sin$ and $\cos$")

# set x limit xlim(-3.0, 3.0) # set y limit ylim(-1.0, 1.0)

# format ticks at specific values
xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],

[r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$']) yticks([-1, 0, +1],
[r'$-1$', r'$0$', r'$+1$'])

show()
```

That should give us a slightly nicer graph:

We see that we used expressions such as $\sin$, or $-\pi$ to write letters of the Greek alphabet in figures. This is LaTex syntax, which we will explore further in the following chapters. Here, we just illustrated how easy it is to make your math charts more readable for certain audiences.

Defining axis lengths and limits

This recipe will demonstrate a variety of useful axis properties around limits and lengths that we can configure in matplotlib.

Getting ready

For this recipe we want to fire up IPython:

$ ipython --pylab

How to do it...

Start experimenting with various properties of axes. Just calling an empty axis() function will return default values for the axis:

In [1]: axis()
Out[1]: (0.0, 1.0, 0.0, 1.0)

Note that if you are in interactive mode and are using a windowing backend, a figure with an empty axis will be displayed.

Here the values represent xmin, xmax, ymin, and ymax respectively. Similarly, we can set values for the x and y axes:

In [2]: l = [-1, 1, -10, 10]
In [3]: axis(l)
Out[3]: [-1, 1, -10, 10]

Again, if you are in interactive mode, this will update the same figure. Furthermore, we can also update any value separately using keyword arguments (**kwargs), setting just xmax to a certain value.

How it works...

If we don't use axis() or other settings, matplotlib will automatically use minimum values that allow us to see all data points on one plot. If we set axis() limits to be less than the maximum values in a dataset, matplotlib will do as told and we will not see all points on the figure. This can be a source of confusion or even error, where we think we see everything we drew. One way to avoid this is to call autoscale() (matplotlib.pyploy.autoscale()), which will compute the optimal size of the axes to fit the data to be displayed.

If we want to add new axes to the same figure, we can use matplotlib.pyploy.axes(). We usually want to add some properties to this default call; for example, rect—which can have the attributes left, bottom, width, and height in normalized units (0, 1)—and maybe axisbg, which specifies the background color of axes.

There are also other properties that we can set for added axes such as

sharex/sharey, which accepts values for other instances of axes and share the current axis (x/y) with other axes. Or parameter polar that defines whether we want to use polar axes.

Adding new axes can be useful; for example, to combine multiple charts on one figure if there is a need to tightly couple different views on the same data to illustrate its properties. If we want to add just one line to the current figure, we can use matplotlib.pyploy. axhline() or matplotlib.pyplot.axvline(). The functions axhilne() and axvline() will draw horizontal and vertical lines across axes for given x and y data values respectively. They share similar parameters, the most important ones being y position, xmin, and xmax for axhline() and x position, ymin, and ymax foraxvline().

Let's see how it looks as a figure, continuing in the same IPython session: **In [3]: axhline()**
Out[3]: <matplotlib.lines.Line2D at 0x414ecd0>
In [4]: axvline()
Out[4]: <matplotlib.lines.Line2D at 0x4152490>
In [5]: axhline(4)
Out[5]: <matplotlib.lines.Line2D at 0x4152850>
We should have a figure like the following plot:

Here we see that just calling these functions without parameters makes them take default values and draw a horizontal line for y=0 (axhline()) and a vertical line for x=0 (axvline()).

Similar to these are two related functions that allow us to add a horizontal span (rectangle) across the axes. These are matplotlib.pyplot.axhspan() and matplotlib.pyplot. axspan(). The function axhspan() has ymin and ymax as required parameters that define how wide the horizontal span is. Analogous to this, axvspan() has xmin and xmax to define the width of the vertical span.

There's more...

Having a grid in a figure is turned off by default, but it can easily be switched on and customized. A default call to matplotlib.pyplot.grid() will toggle the grid's visibility. Other parameters for control are as follows:

f which: This defines what grid tick type to draw (can be major, minor, or both) f

axis: This defines which set of grid lines are drawn (can be both, x, or y)

Axes are usually controlled via matplotlib.pyplot.axis(). Internally, axes are represented by several Python classes, the parent one is matplotlib.axes.Axes, which contains most methods to manipulate axes. A single axis is represented by the matplotlib. axis.Axis class, where the x axis uses matplotlib.axis.XAxis and the y axis uses the matplotlib.axis.YAxis class.

We don't need to use these to perform our recipe, but it is important to know where to look if more advanced axis control interests us and when we hit the limits of what is available via the matplotlib.pyplot namespace.

Defining plot line styles, properties, and format strings

This recipe shows how we can change various line properties, such as styles, colors, or width. Having lines set up appropriately—according to the information presented—and distinct enough for target audiences (if the audience is younger population, we may want to target them with more vivid colors; if they are older we may want to use more contrasting colors) can make the difference between being barely noticeable and leaving a great impact on the viewer.

Getting ready

Although we stressed how important is to aesthetically tune your presentation, we first must learn how to do it.
If you don't have a particular eye for color matching, there are free and commercial online tools that can generate color sets for you. One of the most well known is Colorbrewer2, which can be found at http://colorbrewer2.org/.

Some serious research has been conducted on the usage of color in data visualizations, but explaining that theory is out of the scope of this book. The material on the topic is a must read if you are working with more advanced visualizations daily.

How to do it...

Let us learn how to change line properties. We can change the lines in our plots using different methods and approaches.
First and most common is to define lines by passing keyword parameters to functions such as plot():

plot(x, y, linewidth=1.5)
Because a call to plot() returns the line instance (matplotlib.lines.Line2D), we can use a set of setter methods on that instance to set various properties: line, = plot(x, y)
line.set_linewidth(1.5)
Those who used MATLAB© will feel the need to use a third way of configuring line properties—using the setp() function:
lines = plot(x, y)
setp(lines, 'linewidth', 1.5)

Another way to use setp() is:
setp(lines, linewidth=1.5)

Whatever way you prefer to configure lines, choose one method and stay consistent for the whole project (or at least a file). This way, when you (or someone else that's you in the future) come back to the code, it will be easier to make sense of it and change it.

How it works...

All the properties we can change for a line are contained in the matplotlib.lines. Line2D class. We list some of them in the following table:
Property alpha
Value type float
color or c dashes
Any matplotlib color Sequence of on/off ink in points
label
linestyle or ls
linewidth or lw marker

markeredgecolor or mec
markeredgewidth or mew
markerfacecolor or mfc
markersize or ms solid_capstyle

solid_joinstyle
visible xdata

Any string
['-' | '–' | '-.' | ':' | 'steps' | ...]

float value in points
Description

Sets the alpha value used for blending; not supported on all backends.
Sets the color of the line.

Sets the dash sequence, the sequence of dashes with on/off ink in points. If seq is empty or if seq = (None, None), linestyle will be set tosolid.

Sets the label to s for auto legend. Sets the linestyle of the line
(also accepts drawstyles).

Sets the line width in points.
[7 | 4 | 5 | 6 | 'o' Sets the line marker. | 'D' | 'h' | 'H' |
'_' | '' | 'None' | '
' | None | '8' | 'p'
| ',' | '+' | '.' |
's' | '*' | 'd' | 3
| 0 | 1 | 2 | '1' |
'3' | '4' | '2' | 'v'
| '<' | '>' | '^' |
'|' | 'x' | '$...$' |
tuple | Nx2 array]

Any matplotlib color
Sets the marker edge color.
float value in points Sets the marker edge width in points.
Any matplotlib color Set the marker face color.
float

['butt' | 'round' | 'projecting']
['miter' | 'round' | 'bevel']
[True | False]

np.array

Set the marker size in points. Set the cap style for solid line styles.
Set the join style for solid line styles.

Set the artist's visibility.

Set the data np.array for x. Property Value type ydata np.array Zorder Any number

Description
Set the data np.array for y.
Set the z axis order for the artist.

Artists with lower Zorder values are drawn first.

If x and y are axes going horizontal to the right and vertical to the top of the screen, the z axis is the one extending toward the viewer. So 0 value would be at the screen, 1, one layer above, and so on.

The following table shows some linestyles:

Linestyle
'-'
'--'
'-.'
':'
'None', ' ', ''

Description Solid
Dashed
Dash_dot
Dotted
Draw nothing

The following table shows line markers:

Marker 'o'
'D'
'h'
'H'
'_'
'', 'None', ' ', None
'8'
'p'
','
'+'

'.'
's'
'*'

Description Circle
Diamond
Hexagon1
Hexagon2
Horizontal line Nothing

Octagon Pentagon Pixel
Plus
Point
Square
Star
Marker Description 'd' Thin_diamond 'v' Triangle_down '<' Triangle_left '>' Triangle_right '^' Triangle_up '|' Vertical line 'x' X

Color
We can get all colors that matplotlib supports by calling matplotlib.pyplot.colors(); this will give:

Alias Color b Blue g Green r Red c Cyan m Magenta y Yellow k Black w White

These colors can be used in different matplotlib functions that take color arguments. If these basic colors are not enough—and as we progress, they will not be enough—we can use two other ways of defining a color value. We can use an HTML hexadecimal string: color = '#eeefff'
We can also use legal HTML color names ('red', 'chartreuse'). We can also pass an RGB tuple normalized to [0, 1]:
color = (0.3, 0.3, 0.4)
Argument color is accepted by a range of functions, such as title():
title('Title in a custom color', color='#123456')

Background color
By providing axisbg to a function such as matplotlib.pyplot.axes() or matplotlib.pyplot.subplot(), we can define the background color of an axis:

subplot(111, axisbg=(0.1843, 0.3098, 0.3098))

Setting ticks, labels, and grids

In this recipe we will continue with setting axis and line properties and adding more data to our figure and charts.

Getting ready

Let us learn a little about figures and subplots.

In matplotlib, figure() is used to explicitly create a figure, which represents a user interface window. Figures are created implicitly just by calling plot() or similar functions. This is fine for simple charts, but having the ability to explicitly create a figure and get a reference to its instance is very useful for more advanced use.

A figure contains one or more subplots. Subplots allow us to arrange plots in a regular grid. We already used subplot(), in which we specify the number of rows and columns and the number of the plot we are referring to.

If we want more control, we need to use axes instances from the matplotlib.axes.Axes class. They allow us to place plots at any location in the figure. An example of this would be to put a smaller plot inside a bigger one.

How to do it...

Ticks are part of figures. They consist of tick locators—where ticks appear—and tick formatters which show how ticks appear. There are major and minor ticks. Minor ticks are not visible by default. More importantly, major and minor ticks can be formatted and located independently of each other.

We can use matplotlib.pyplot.locator_params() to control the behavior of tick locators. Even though tick locations are usually determined automatically, we can control the number of ticks and use a tight view if we want to, for when plots are smaller.

```
from pylab import *

# get current axis ax = gca()
# set view to tight, and maximum number of tick intervals to 10
ax.locator_params(tight=True, nbins = 10)

# generate 100 normal distribution values ax.plot(np.random.normal(10, .1,
```

100))
show()
This should give us the following graph:

We see how the x and y axes are divided and what values are shown. We could have achieved the same setup using locator classes. Here we are saying "set the major locator to be a multiple of 10":

ax.xaxis.set_major_locator(matplotlib.ticker.MultipleLocator(10))

Tick formatters can similarly be specified. Formatters specify how the values (usually numbers) are displayed. For example, matplotlib.ticker.FormatStrFormatter simply specifies '%2.1f' or '%1.1f cm' as the string to be used as the label for the ticker.

Let's take a look at one example using dates.
matplotlib represents dates in floating point values as the time in days passed since 0001-01-01 UTC plus 1. So, 0001-01-01 UTC 06:00 is 1.25.

Then we can use helper functions such as matplotlib.dates.date2num(), matplotlib.dates.num2date(), and matplotlib.dates.drange() to convert dates between different representations.

Let's see another example:

```
from pylab import *
import matplotlib as mpl import datetime

fig = figure()
# get current axis ax = gca()

# set some daterange
start = datetime.datetime(2013, 01, 01) stop = datetime.datetime(2013, 12, 31)
delta = datetime.timedelta(days = 1)

# convert dates for matplotlib
dates = mpl.dates.drange(start, stop, delta)
# generate some random values values = np.random.rand(len(dates))
ax = gca()
# create plot with dates
ax.plot_date(dates, values, linestyle='-', marker='')
# specify formater
date_format = mpl.dates.DateFormatter('%Y-%m-%d')
# apply formater
ax.xaxis.set_major_formatter(date_format)

# autoformat date labels
# rotates labels by 30 degrees by default
# use rotate param to specify different rotation degree # use bottom param to
give more room to date labels fig.autofmt_xdate()

show()
```

The preceding code will give us the following graph:

Adding a legend and annotations

Legends and annotations explain data plots clearly and in context. By assigning each plot a short description about what data it represents, we are enabling an easier mental model in the reader's (viewer's) head. This recipe will show how to annotate specific points on our figures and how to create and position data legends.

Getting ready

How many times have you looked at a chart and wondered what the data represents? More often than not, newspapers and other daily and weekly publications create plots that don't contain appropriate legends, thus leaving the reader free to interpret the representation. This creates ambiguity for the readers and increases the possibility of error.

How to do it...

Let us demonstrate how to add legends and annotations with the following example:

```
from matplotlib.pyplot import *

# generate different normal distributions
x1 = np.random.normal(30, 3, 100)
```

```
x2 = np.random.normal(20, 2, 100) x3 = np.random.normal(10, 3, 100)

# plot them
plot(x1, label='plot') plot(x2, label='2nd plot') plot(x3, label='last plot')

# generate a legend box
legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3, ncol=3, mode="expand",
borderaxespad=0.)

# annotate an important value
annotate("Important value", (55,20), xycoords='data', xytext=(5, 38),
arrowprops=dict(arrowstyle='->'))
show()
```

The preceding code will give us the following plot:

What we do is assign a string label with every plot so legend() will try and determine what to add in the legend box.
We set the location of a legend box by defining the loc parameter. This is optional, but we want to specify a location where it is least likely for the legend box to be drawn over plot lines.

How it works...

All location parameter strings are given in the following table:

String Number value upper right 1
upper left 2
lower left 3
lower right 4
right 5
center left 6
center right 7
lower center 8
upper center 9
center 10

To not show the label in a legend, set the label to _nolegend_.

For the legend, we defined the number of columns with ncol = 3 and set the location with lower left. We specified a bounding box (bbox_to_anchor) to start from position (0., 1.02), and to have a width of 1 and a height of 0.102. These are normalized axis coordinates. Parameter mode is either None or expand to allow the legend box to expand horizontally filling the axis area. The parameter borderaxespad defines the padding between the axes and the legend border.

For annotations, we have defined a string to be drawn on a plot, on a coordinate xy. The coordinate system is specified to be the same as the data one; therefore, coordinate system is xycoord = 'data'. The starting position for the text is defined by the value of xytext.

An arrow is drawn from xytext toxy coordinate and the arrowprops dictionary can define many properties for that arrow. For this example, we used arrowstyle to define arrow style.

Moving spines to the center

This recipe will demonstrate how to move spines to the center.

Spines define data area boundaries; they connect the axis tick marks. There are four spines. We can place them wherever we want; by default, they are placed on the border of the axis, hence we see a box around our data plot.

How to do it...

To move the spines to the center of the plot, we need to remove two spines, making them hidden (set color tonone). After that we move two others to coordinate (0,0). The coordinates are specified in data space coordinates.

The following code shows how to do this:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi, np.pi, 500, endpoint=True) y = np.sin(x)
plt.plot(x, y)
ax = plt.gca()

# hide two spines
ax.spines['right'].set_color('none') ax.spines['top'].set_color('none')

# move bottom and left spine to 0,0
ax.spines['bottom'].set_position(('data',0)) ax.spines['left'].set_position(('data',0))

# move ticks positions
ax.xaxis.set_ticks_position('bottom') ax.yaxis.set_ticks_position('left')

plt.show()
```

This is what the plot will look like:

How it works...

This code is dependent on the plot that is drawn because we are moving spines to the location (0, 0) and are plotting a sine function on the interval where (0, 0) is in the middle of the plot. Nevertheless, this demonstrated how to move spines to a particular location and how to get rid of spines we don't want to show.

There's more...

Furthermore, spines can be limited to end where the data ends, for example, using a set_smart_bounds(True) call. In this case matplotlib tries to set bounds in a sophisticated way, for example, to handle inverted limits, or to clip line to view if data extends past view.

Making histograms

Histograms are simple, yet it's important to get the right data into them. We will cover histograms in 2D for now.

Histograms are used to visualize estimations of distribution of data. Generally, we use a few terms when speaking of histograms. Vertical rectangles represent frequencies of data points within a particular interval, called a bin. Bins are created at fixed intervals so the total area of a histogram equals the number of data points.

Instead of using absolute values of data, histograms can display relative frequencies of data. When this is the case, the total area equals 1.

Histograms are often used in image manipulation software as a way to visualize image properties, such as distribution of light in a particular color channel. Further, these image histograms can be used in computer vision algorithms to detect peaks aiding in edge detections, image segmentation, and so on.

In *Chapter 5*, *Making 3D Visualizations*, we have recipes that deal with 3D histograms.

Getting ready

The number of bins is the value we want to get right, but it is hard to get right as there are no strict rules on what is the optimal number of bins. There are different theories on how to calculate number of bins, the simplest being the one based on a ceiling function, where the number of bins (*k*) is equal to the ceiling *(max(x) – min(x)/h)* where *x* is the dataset plotted, and *h* is the desired bin width. This is just one option as the number of bins required to display data properly is dependent on real data distribution.

How to do it...

We create a histogram calling matplotlib.pyploy.hist() with a set of parameters. Here are some of the most useful ones:

f bins: This is either an integer number of bins or a sequence giving the bins.

The default is 10.

f range: This is the range of bins and is not used if bins are given as a sequence. Outliers are ignored and the default is None.

f normed: If the value for this is True, histogram values are normalized and form probability density. The default is False.

f histtype: This is the default bar-type histogram. The other options are: • barstacked: This for multiple data gives stacked-view histograms. • step: This creates a line plot that is left unfilled.

• stepfilled: This creates line plot that is filled by default. The default is bar.

f align: This centers bars between bin edges. The default is mid. Other values are left and right.

f color: This specifies the color of the histogram. It may be a single value or have a sequence of colors. If multiple datasets are specified, the color sequence

will be used in the same order. If not specified, a default line color sequence is used.

f orientation: This allows the creation of histograms that are horizontal by setting orientation tohorizontal. The default is vertical.
The following code demonstrates how hist() is used:

```
import numpy as np
import matplotlib.pyplot as plt

mu = 100
sigma = 15
x = np.random.normal(mu, sigma, 10000)

ax = plt.gca()
# the histogram of the data
ax.hist(x, bins=35, color='r')
ax.set_xlabel('Values')
ax.set_ylabel('Frequency')
ax.set_title(r'$\mathrm{Histogram:}\ \mu=%d,\ \sigma=%d$' % (mu, sigma))
plt.show()
```

This creates a neat, red-colored histogram for our data sample:

How it works...

We start by generating some normally distributed data. The histogram is plotted with the specified number of bins—35—and it is normalized by setting normed toTrue (or 1); we set the color tored (r).

After that, we set labels and a title for the plot. Here we used the ability to write LaTeX expressions to write math symbols and mixed that with Python format strings.

Making bar charts with error bars

In this recipe, we will show how to create bar charts and how to draw error bars.

Getting ready

To visualize uncertainty of measurement in our dataset or to indicate the error, we can use error bars. Error bars can easily give an idea of how error free the dataset is. They can show one standard deviation, one standard error, or 95 percent confidence interval. There is no standard here, so always explicitly state what values (errors) error bars display. Most papers in the experimental sciences should contain error bars to present accuracy of the data.

How to do it...

Even though just two parameters are mandatory—left and height— we often want to use more than that. Here are some parameters we can use:

f width: This gives the width of the bars. The default value is 0.8.
f bottom: If bottom is specified, the value is added to the height. The default is None. f edgecolor: This gives the color of the bar edges.
f ecolor: This specifies the color of any error bar.

f linewidth: This gives width of bar edges; special values areNone (use defaults) and 0 (when bar edges are not displayed).
f orientation: This has two values vertical and horizontal.
f xerr and yerr: These are used to generate error bars on the bar chart. Some optional arguments (color, edgecolor, linewidth, xerr, and yerr) can be single values or sequences with the same length as the number of bars.

How it works...

Let us illustrate this using an example:

```
import numpy as np
import matplotlib.pyplot as plt
# generate number of measurements
x = np.arange(0, 10, 1)
# values computed from "measured"
y = np.log(x)
# add some error samples from standard normal distribution xe = 0.1 *
np.abs(np.random.randn(len(y)))

# draw and show errorbar
plt.bar(x, y, yerr=xe, width=0.4, align='center', ecolor='r', color='cyan',
label='experiment #1');

# give some explainations
plt.xlabel('# measurement')
plt.ylabel('Measured values')
plt.title('Measurements')
plt.legend(loc='upper left')

plt.show()
```

The preceding code will plot the following:

To be able to plot an error bar, we needed to have some measures (x); for every measure computed (y), we introduced errors (xe).

We used NumPy to generate and compute values; standard distributions are good enough for demonstration purposes, but if you happen to know your data distribution in advance, you can always make some prototype visualizations and try out different layouts to find the best options to present information.

Another interesting option to use if we are preparing visualizations for a black-and-white medium is hatch; it can have the following values:

Hatch value /
\
|
-
+
x
o
0
.
*

Description
Diagonal hatching Back diagonal
Vertical hatching Horizontal
Crossed
Crossed diagonal Small circle
Large circle
Dot pattern
Star pattern

There's more...

What we have just used are error bars known as symmetrical error bars. If the nature of our dataset is such that errors are not the same in both directions (negative and positive), we can also specify them separately using asymmetrical error bars.

All we have to do differently is to specify xerr or yerr using a two-element list (such as a 2D array), where the first list contains values for negative errors and

the second one for positive errors.

Making pie charts count

Pie charts are special in many ways, the most important being that the dataset they display must sum up to 100 percent or they are just plain not valid.

Getting ready

Pie charts represent numerical proportions, where the arc length of each segment is proportional to the quantity it represents.

They are compact, can look very aesthetically pleasing, but they have been criticized as they can be hard to compare. Another property of pie charts that does not work in their best interest is that pie charts are presented in a specific angle (perspective)—and segments use certain colors—that can skew our perception and influence our conclusion about information presented.

What we will show here is different ways to use pie charts to present data.

How to do it...

As a start, we create a so-called exploded pie chart:

```
from pylab import *

# make a square figure and axes figure(1, figsize=(6,6))
ax = axes([0.1, 0.1, 0.8, 0.8])

# the slices will be ordered
# and plotted counter-clockwise.
labels = 'Spring', 'Summer', 'Autumn', 'Winter'

# fractions are either x/sum(x) or x if sum(x) <= 1 x = [15, 30, 45, 10]
# explode must be len(x) sequence or None explode=(0.1, 0.1, 0.1, 0.1)
pie(x, explode=explode, labels=labels, autopct='%1.1f%%', startangle=67)
title('Rainy days by season')
show()
```

Pie charts look best if they are inside a square figure and have square axes.

Fractions of the whole sum of the pie chart are defined as x/sum(x), or x if sum(x) <= 1. We get the explode effect by defining an explode sequence where each item represents the fraction of radius with which to offset each arc. We use the autopct parameter to format the labels that will be drawn inside the arcs; they

can be a format string or a callable (function).

We can also use a Boolean shadow parameter to add a shadow effect to a pie chart.

If we don't specify startangle, the fractions will be ordered starting counterclockwise from the x axis (angle 0). If we specify 90 as the value of startangle, that will start the pie chart from the y axis.

This is the resulting pie chart:

Plotting with filled areas

In this recipe, we will show you how to fill the area under a curve or in between two different curves.

Getting ready

Library matplotlib allows us to fill areas in between and under the curves with color so that we can display the value of that area to the spectator. Sometimes, it is necessary for readers (viewers) to comprehend the given specialization.

How to do it...

Here's one example of how to fill areas between two contours:

```
from matplotlib.pyplot import figure, show, gca
import numpy as np
x = np.arange(0.0, 2, 0.01)

# two different signals are measured
y1 = np.sin(2*np.pi*x)
y2 = 1.2*np.sin(4*np.pi*x)

fig = figure()
ax = gca()

# plot and
# fill between y1 and y2 where a logical condition is met ax.plot(x, y1, x, y2,
color='black')

ax.fill_between(x, y1, y2, where=y2>=y1, facecolor='darkblue',
interpolate=True)
ax.fill_between(x, y1, y2, where=y2<=y1, facecolor='deeppink',
interpolate=True)

ax.set_title('filled between')
show()
```

How it works...

After we have generated random signals for a predefined interval, we plot these two signals using a regular plot(). Then we call fill_between() with properties that are required and mandatory.

The function fill_between() is using x as the location from where to pick y values (y1, y2) and will then plot the polygon in certain defined colors.
We specify a condition to fill the curve with the where parameter, which accepts Boolean values (can be expressions) so that the fill happens only when the where condition is met.

There's more...

Similar to other functions for plotting, this function also accepts many more parameters, for example, hatch (to specify patterns to fill with instead of color), and line options (linewidth and linestyle).

There is also fill_betweenx(), which enables similar fill features, but it does so between horizontal curves.
The more general function fill() provides the ability to fill any polygon with a color or a hatch.

Drawing scatter plots with colored markers

If you have two variables and want to spot the correlation between those, a scatter plot may be the solution to spot patterns.
This type of plot is also very usable as a start for more advanced visualization of multidimensional data, for example, to plot a scatter plot matrix.

Getting ready

Scatter plots display values for two sets of data. The data visualization is done as a collection of points not connected by lines. Each of them has its coordinates determined by the value of the variables. One variable is controlled (independent variable) while the other variable is measured (dependent variable) and is often

plotted on the y axis.

How to do it...

Here's a code sample that plots two plots: one with uncorrelated data and other with strong positive correlation:

```
import matplotlib.pyplot as plt
import numpy as np
# generate x values
x = np.random.randn(1000)
# random measurements, no correlation y1 = np.random.randn(len(x))
# strong correlation y2 = 1.2 + np.exp(x)

ax1 = plt.subplot(121)
plt.scatter(x, y1, color='indigo', alpha=0.3, edgecolors='white', label='no correl')
plt.xlabel('no correlation')
plt.grid(True)
plt.legend()

ax2 = plt.subplot(122, sharey=ax1, sharex=ax1)
plt.scatter(x, y2, color='green', alpha=0.3, edgecolors='grey', label='correl')
plt.xlabel('strong correlation')
plt.grid(True)
plt.legend()

plt.show()
```

Here, we also use more parameters, such as color for setting the color of the plot, marker for using as a point marker (the default is circle), alpha (alpha transparency), edgecolors (color of the marker edge), and label (for legend box).

These are the plots we get:

How it works...

A scatter plot is often used to identify potential association between two variables, and it's often drawn before working on a fitting regression function. It gives a good visual picture of the correlation, particularly for nonlinear relationships. matplotlib provides the scatter() function to plot x versus y unidimensional array of the same length as a scatter plot.

4

More Plots and Customizations

In this chapter we will learn about:
f Setting the transparency and size of axis labels

f Adding a shadow to the chart line
f Adding a data table to the figure
f Using subplots
f Customizing grids
f Creating contour plots
f Filling an under-plot area
f Drawing polar plots
f Visualizing the filesystem tree using a polar bar

Introduction

In this chapter we will explore more advanced properties of the matplolib library. We are going to introduce more options and will look at how to achieve certain visually pleasing results.

During this chapter we will seek the solutions to some non-trivial problems with representing data when simple charts are not enough. We will try to use more than one type of graph or create hybrid ones to cover some advanced data structures and the representation required.

Setting the transparency and size of axis labels

The Axes label describes what the data in the figure represents and is quite important in the viewer's understanding of the figure itself. By providing labels to the axes background, we help the viewer comprehend the information in an appropriate way.

Getting ready

Before we dive into the code, it is important to understand how matplotlib

organizes our figures.

At the top level, there is a Figure instance containing all that we see and some more (that we don't see). The figure contains, among other things, instances of the Axes class as a field Figure.axes. The Axes instances contain almost everything we care about: all the lines, points, and ticks and labels. So, when we call plot(), we are adding a line (matplotlib. lines.Line2D) to the Axes.lines list. If we plot a histogram (hist()), we are adding rectangles to the list of Axes.patches ("patches" is the term inherited from MATLAB™, and represents the "patch of color" concept).

An instance of Axes also holds references to the XAxis and YAxis instances, which in turn refer to the x axis and y axis, respectively. XAxis and YAxis manage the drawing of the axis, labels, ticks, tick labels, locators, and formatters. We can reference those through Axes. xaxis and Axes.yaxis, respectively. We don't have to go all the way down to XAxis or YAxis instances to get to the labels as matplotlib gives us a helper method (practically a shortcut) that enables iterations via these labels: matplotlib.pyplot.xlabel() and matplotlib.pyplot.ylabel().

How to do it...

We will now create a new figure where we will:

1. Create a plot with some random generated data.
2. Add the title and axes labels.
3. Add alpha settings.
4. Add shadow effects to the title and axes labels.

```
import matplotlib.pyplot as plt from matplotlib import patheffects import numpy as np
data = np.random.randn(70)

fontsize = 18 plt.plot(data)

title = "This is figure title" x_label = "This is x axis label" y_label = "This is y axis label"

title_text_obj = plt.title(title, fontsize=fontsize, verticalalignment='bottom')
title_text_obj.set_path_effects([patheffects. withSimplePatchShadow()])
```

offset_xy -- set the 'angle' of the shadow # shadow_rgbFace -- set the color of the shadow # patch_alpha -- setup the transparency of the shadow

offset_xy = (1, -1) rgbRed = (1.0,0.0,0.0) alpha = 0.8

```
# customize shadow properties
pe = patheffects.withSimplePatchShadow(offset_xy = offset_xy,
shadow_rgbFace = rgbRed,
patch_alpha = alpha) # apply them to the xaxis and yaxis labels
xlabel_obj = plt.xlabel(x_label, fontsize=fontsize, alpha=0.5)
xlabel_obj.set_path_effects([pe])

ylabel_obj = plt.ylabel(y_label, fontsize=fontsize, alpha=0.5)
ylabel_obj.set_path_effects([pe])
plt.show()
```

How it works...

We already know all the familiar imports, parts that generate data, and basic plotting techniques so we will skip that. If you are not able to decipher the first few lines of the example, please refer to *Chapter 2*, *Knowing Your Data*, and *Chapter 3*, *Drawing Your First Plots and Customizing Them*, where these concepts are already explained. After we plot the dataset, we are ready to add titles and labels, and customize their appearance.

First, we add the title. Then we define the font size and vertical alignment of the title text to be bottom. The default shadow effect is added to the title if we are using matplotlib. patheffects.withSimplePatchShadow() with no parameters. The default values for parameters are: offset_xy=(2,-2), shadow_rgbFace=None, and patch_alpha=0.7. The other values are center, top, and baseline but we choose bottom as the text will have some shadow. In the next line, we add shadow effect. The path effects are part of the matplotlib module matplotlib.patheffects that supports matplotlib.text.Text and matplotlib.patches.Patch.

We now want to add different settings of the shadow to both the x and y axes. First, we customize the position (offset) of the shadow to the parent object, and then we set the color of the shadow. The color is here represented in triples (3-tuple) of float values between 0.0 and 1.0, for each of the RGB channels. Therefore, our red color is represented as (1.0, 0.0, 0.0) (all red, no green, no

blue).

The transparency (or alpha) is set up as a normalized value, and we also want to set this up here to be different from default.

With all the settings there, we instantiate matplotlib.patheffects.withSimplePatchShadow and hold the reference to it in the variable pe to reuse it few lines later.

To be able to apply the shadow effect, we need to get to the label object. This is simple enough because matplotlib.pyplot.xlabel() returns a reference to the object (matplotlib.text.Text) that we use then to call set_path_effects([pe]).

We finally show the plot and feel proud of our work.

There's more...

If you are not satisfied with the effects that matplotlib.patheffects currently offers, you can inherit the matplotlib.patheffects._Base class and override the draw_path method. Take a look at the code and comments on how to do that here:

https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/patheffects.py#L47

Adding a shadow to the chart line

To be able to distinguish one particular plot line in the figure or just to fit in the overall style of the output our figure is in, we sometimes need to add a shadow effect to the chart line (or histogram, for that matter). In this recipe we will be learning how to add a shadow effect to the plot's chart lines.

Getting ready

To add shadows to the lines or rectangles in our charts, we need to use the transformation framework built in matplotlib and located in matplotlib.transforms. To understand how it all works, we need to explain what transformations are in matplotlib and how they work.

Transformations know how to convert the given coordinates from their coordinate system into display. They also know how to convert them from display coordinates into their own coordinate system.

The following table summarizes existing coordinate systems and what they represent:

Coordinate system
Data
Axes

Transformation object Description
Axes.transData Axes.transAxes
Figure Figure.transFigure
Display None
Represents the user's data coordinate system.

Represents the Axes coordinate system, where (0,0) represents the bottom-left end of the axes and (1,1) represents the upper-right end of the axes.

This is the Figure coordinate system, where (0,0) represents the bottom-left end of the figure and (1,1) represents the upper-right end of the figure.

Represents the pixel coordinate system of the user display, where (0,0) represents the bottom-left of the display, and tuple (width, height) represents the upper-right of the display, where width and height are in pixels.

Note how the display does not have a value in the column. This is because the default coordinate system is Display, so coordinates are always in pixels relative to your display coordinate systems. This is not very useful, and most often we want them normalized into Figure or Axes or a Data coordinate system.
This framework enables us to transform the current object into an offset object, that is, to place that object shifted for a certain distance from the original object.

We will use this framework to create our desired effect on the plotted sine wave.
How to do it...
Here is the code recipe to add shadow to the plotted chart. The code is explained in the section that follows.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.transforms as transforms

def setup(layout=None):
```

```
assert layout is not None

fig = plt.figure()
ax = fig.add_subplot(layout) return fig, ax

def get_signal():
t = np.arange(0., 2.5, 0.01) s = np.sin(5 * np.pi * t) return t, s

def plot_signal(t, s):
line, = axes.plot(t, s, linewidth=5, color='magenta') return line,

def make_shadow(fig, axes, line, t, s):
delta = 2 / 72. # how many points to move the shadow offset =
transforms.ScaledTranslation(delta, -delta, fig.dpi_

scale_trans)
offset_transform = axes.transData + offset

# We plot the same data, but now using offset transform # zorder -- to render it
below the line
axes.plot(t, s, linewidth=5, color='gray',

transform=offset_transform,
zorder=0.5 * line.get_zorder())

if __name__ == "__main__": fig, axes = setup(111) t, s = get_signal()
line, = plot_signal(t, s)

make_shadow(fig, axes, line, t, s)
axes.set_title('Shadow effect using an offset transform') plt.show()
```

How it works...

We start reading the code from the bottom, after the if __name__ check. First, we create the figure and axes in setup(); after that, we obtain a signal (or generate data—sine wave). We plot the basic signal in plot_signal(). Then, we make the shadow transformation and plot the shadow in make_shadow().

We use the offset effect to create an offset object underneath and just few points away from original object.

The original object is a simple sine wave that we plot using the standard function plot(). To add to this offset transformation, matplotlib contains helper transformation— matplotlib.transforms.ScaledTranslation.
The values for dx and dy are defined in points, and as the point is 1/72 inches, we move the offset object 2pt right and 2pt down.

If you want to learn more about how we converted the point to 1/71 inches, read more in this Wikipedia article: http://en.wikipedia.org/wiki/Point_%28typography%29.

We can use matplotlib.transforms.ScaledTransformation(xtr, ytr, scaletr); here, xtr and ytr are translation offsets and scaletr is a transformation callable to scale xtr and ytr at transformation time and before display. The most common use case for this is transforming from points to display space: for example, to DPI so that the offset always stays at the same place no matter what the actual output—be it the monitor or printed material. The callable we use for this is already built in, and is available at Figure.dpi_scale_trans.

We then plot the same data with the applied transformation.

There's more...

Using transforms to add shadows is just one and not the most popular use case of this framework. To be able to do more with transformation framework, you will need to learn the details of how the transformation pipeline works and what the extension points are (what classes to inherit and how). It's easy enough because matplotlib is open source, and even if some code is not well documented, there is a source you can read from and use or change, thus contributing to the overall quality and usefulness of matplotlib.

Adding a data table to the figure

Although matplotlib is mainly a plotting library, it helps us with small errands when we are creating a chart, such as having a neat data table beside our beautiful chart. In this recipe we will be learning how to display a data table alongside the plots in the figure.

Getting ready

It is important to understand why we are adding a table to a chart. The main

intention of plotting data visually is to explain the otherwise not understandable (or hardly understandable) data values. Now, we want to add that data back. It is not wise just to cram a big table with values underneath the chart.

But, carefully picked, maybe the summed or highlighted values from the whole, charted dataset can identify important parts of the chart or emphasize the important values for those places where the exact value (for example, yearly sales in USD) is important (or even required).

How to do it...

Here's the code to add a sample table to our figure:

```
import matplotlib.pylab as plt
import numpy as np

plt.figure()
ax = plt.gca()
y = np.random.randn(9)

col_labels = ['col1','col2','col3']
row_labels = ['row1','row2','row3']
table_vals = [[11, 12, 13], [21, 22, 23], [28, 29, 30]] row_colors = ['red', 'gold',
'green']
my_table = plt.table(cellText=table_vals,

colWidths=[0.1] * 3, rowLabels=row_labels, colLabels=col_labels,
rowColours=row_colors, loc='upper right')

plt.plot(y) plt.show()
```

The previous code snippet gives a plot such as the following:

How it works...

Using plt.table() we create a table of cells and add it to the current axes. The table can have (optional) row and column headers. Each table cell contains either patch or text. The column widths and row heights for the table can be specified. The return value is a sequence of objects (text, line, and patch instances) that the table is made of.
The basic function signature is:

table(cellText=None, cellColours=None,
cellLoc='right', colWidths=None,
rowLabels=None, rowColours=None, rowLoc='left', colLabels=None,
colColours=None, colLoc='center', loc='bottom', bbox=None)

The function instantiates and returns the matplotlib.table.Table instance. This is usually the case with matplotlib; there's just one way to add the table to the figure. The object-oriented interface can be directly accessed. We can use the matplotlib.table. Table class directly to fine-tune our table before we add it onto our axes instance with add_table().

There's more...

You can have more control if you directly create an instance of

matplotlib.table.Table and configure it before you add it to the axes instance. You can add the table instance toaxes using Axes.add_table(table), where table is an instance of matplotlib. table.Table.

Using subplots

If you are reading this book from the start, you are probably familiar with the subplot class, a descendant of axes that lives on the regular grid of subplot instances. We are going to explain and demonstrate how to use subplots in advanced ways.

In this recipe we will be learning how to create custom subplot configurations on our plots.

Getting ready

The base class for subplots is matplotlib.axes.SubplotBase. These subplots are matplotlib.axes.Axes instances but provides helper methods for generating and manipulating a set of Axes within a figure.

There is a class matplotlib.figure.SubplotParams, which holds all the parameters for subplot. The dimensions are normalized to the width or height of the figure. As we already know, if we don't specify any custom values, they will be read from the rc parameters.

The scripting layer (matplotlib.pyplot) holds a few helper methods to manipulate subplots.

matplotlib.pyplot.subplots is used for the easy creation of common layouts of subplots. We can specify the size of the grid—the number of rows and columns of the subplot grid.
We can create subplots that share the x or y axes. This is achieved using sharex or the sharey keyword argument. The argument sharex can have the value True, in which case the x axis is shared among all the subplots. The tick labels will be invisible on all but the last row of plots. They can also be defined as String, with enumerated values of row, col, all, or none. The value all is the same as True, and the value none is the same as False. If the value row is specified, each subplot row shares the x axis. If the value col is specified, each subplot column shares the x axis. This helper returns tuple fig, ax where ax is either an axis instance or, if more than one subplot is created, an array of axis instances.

matplotlib.pyplot.subplots_adjust is used to tune the subplot layout. The keyword arguments specify the coordinates of the subplots inside the figure (left, right, bottom, and top) normalized to figure size. White space can be specified to be left between the subplots using the wspace and hspace arguments for width and height amounts respectively.

How to do it...

1. We will show you an example of using yet another helper function in the matplotlib toolkit—subplot2grid. We define the grid's geometry and the subplot location. Note that this location is 0-based, not 1-based as we are used to in plot.subplot(). We can also use colspan and rowspan to allow the subplot to span multiple columns and rows in a given grid. For example, we will: create a figure; add various subplot layouts using subplot2grid; reconfigure the tick label size.

2. Show the plot:

```
import matplotlib.pyplot as plt

plt.figure(0)
axes1 = plt.subplot2grid((3, 3), (0, 0), colspan=3) axes2 = plt.subplot2grid((3,
3), (1, 0), colspan=2) axes3 = plt.subplot2grid((3, 3), (1, 2))
axes4 = plt.subplot2grid((3, 3), (2, 0))
axes5 = plt.subplot2grid((3, 3), (2, 1), colspan=2)

# tidy up tick labels size
all_axes = plt.gcf().axes
for ax in all_axes:

for ticklabel in ax.get_xticklabels() + ax.get_yticklabels():
ticklabel.set_fontsize(10)
plt.suptitle("Demo of subplot2grid") plt.show()
```

When we execute the previous code, the following plot is created:

How it works...

We provide subplot2grid with a shape, location (loc), and optionally, rowspan and colspan. The important difference here is that the location is indexed from 0, and not from 1, as in figure.add_subplot.

There's more...

To give an example of another way you can customize the current axes or subplot:

```
axes = fig.add_subplot(111)
rectangle = axes.patch
rectangle.set_facecolor('blue')
```

Here we see that every axes instance contains a field patch referencing the rectangle instance, thus representing the background of the current axes instance. This instance has properties that we can update, hence updating the current axes background. We can change its color, but we can also load an image to add a watermark protection, for example. It is also possible to create a patch first and then just add it to the axes background:

```
fig = plt.figure()
```

```
axes = fig.add_subplot(111)
rect = matplotlib.patches.Rectangle((1,1), width=6, height=12)
axes.add_patch(rect)
# we have to manually force a figure draw
axes.figure.canvas.draw()
```

Customizing grids

A grid is usually handy to have under lines and charts as it helps the human eye spot differences in pattern and compare plots visually in the figure. To be able to set up how visibly, how frequently, and in what style the grid is displayed—or whether it is displayed at all—we should use matplotlib.pyplot.grid.

In this recipe we will be learning how to turn the grid on and off, and how to change the major and minor ticks on a grid.

Getting ready

The most frequent grid customization is reachable in the matplotlib.pyplot.grid helper function.

To see the interactive effect of this, you should run the following under ipython –pylab. The basic call to plt.grid() will toggle grid visibility in the current interactive session started by the last IPython PyLab environment:

```
In [1]: plt.plot([1,2,3,3.5,4,4.3,3])
Out[1]: [<matplotlib.lines.Line2D at 0x3dcc810>]
```

Now we can toggle the grid on the same figure:
In [2]: plt.grid()
We turn the grid back on, as shown in the following plot:

And then we

turn if off again:
In [3]: plt.grid()

Apart from just turning them on and off, we can further customize the grid appearance. We can manipulate the grid with just major ticks, or just minor ticks, or both; hence, the value of function argument which can be 'major', 'minor', or 'both'. Similar to this, we can control the horizontal and vertical ticks separately by using the argument axis that can have values 'x', 'y', or 'both'.

All the other properties are passed via kwargs and represent a standard set of properties that a matplotlib.lines.Line2D instance can accept, such as color, linestyle, and linewidth; here is an example:

ax.grid(color='g', linestyle='--', linewidth=1)
How to do it...

This is nice, but we want to be able to customize more. In order to do that we need to reach deeper into matplotlib and into mpl_toolkits and find the AxesGrid module that allows us to make grids of axes in an easy and manageable way:

import numpy as np
import matplotlib.pyplot as plt

```
from mpl_toolkits.axes_grid1 import ImageGrid
from matplotlib.cbook import get_sample_data

def get_demo_image():
f = get_sample_data("axes_grid/bivariate_normal.npy",
asfileobj=False)
# z is a numpy array of 15x15
Z = np.load(f)
return Z, (-3, 4, -4, 3)

def get_grid(fig=None, layout=None, nrows_ncols=None): assert fig is not None
assert layout is not None
assert nrows_ncols is not None

grid = ImageGrid(fig, layout, nrows_ncols=nrows_ncols, axes_pad=0.05,
add_all=True, label_mode="L")
return grid

def load_images_to_grid(grid, Z, *images):
min, max = Z.min(), Z.max()
for i, image in enumerate(images):

axes = grid[i]
axes.imshow(image, origin="lower", vmin=min, vmax=max,
interpolation="nearest")

if __name__ == "__main__":
fig = plt.figure(1, (8, 6)) grid = get_grid(fig, 111, (1, 3)) Z, extent =
get_demo_image()

# Slice image
image1 = Z
image2 = Z[:, :10] image3 = Z[:, 10:]

load_images_to_grid(grid, Z, image1, image2, image3)
plt.draw() plt.show()
```

The given code will render the following plot:

How it works...

In the function get_demo_image, we loaded data from the sample data directory that comes with matplotlib.
The list grid holds our axes grid (in this case, ImageGrid).

The variables image1, image2, and image3 hold sliced data from Z that we have split over multiple axes in the list grid.
Looping over all the grids, we are plotting data from im1, im2, and im3 using the standard imshow() call, while matplotlib takes care that everything is neatly rendered and aligned.

Creating contour plots

A contour plot displays the isolines of matrix. Isolines are curves where a function of two variables has the same value.
In this recipe we will learn how to create contour plots.

Getting ready

Contours are represented as a contour plot of matrix Z, where Z is interpreted as height with respect to the X-Y plane. Z is of minimum size 2 and must contain at least two different values.

The problem with contour plots is that if they are coded without labeling the isolines, they render pretty useless as we cannot decode the high points from low points or find local minimas.

Here we need to label the contour also. The labeling of isolines can be done either by using labels (clabel()) or colormaps. If your output medium permits the

usage of color, colormaps are preferred because viewers will be able to decode data more easily.

The other risk with contour plots is choosing the number of isolines to plot. If we choose too many, the plot becomes too dense to decode, and if we go with too few isolines, we lose information and can perceive data differently.

The function contour() will automatically guess how many isolines to plot, but we also have the ability to specify our own number.
In matplotlib, we draw contour plots using matplotlib.pyplot.contour.

There are two similar functions: contour() draws contour lines, and contourf() draws filled contours. We are going to demonstrate only contour(), but almost everything is applicable to contourf(). They understand almost the same arguments as well. The function contour() can have different call signatures, depending on what data we have and/or what the properties that we want to visualize are.

Call signature
contour(Z)
contour(X,Y,Z)

contour(Z,N)
contour(X,Y,Z,N)

contour(Z,V)
contour(X,Y,Z,V) contourf(..., V)

contour(Z, **kwargs)

Description
Plots the contour of Z (array). The level values are chosen automatically.
Plots the contour of X, Y, and Z. The arrays X and Y are (x, y) surface coordinates.
Plots the contour of Z, where the number of levels is defined with N. The level values are automatically chosen. Plots the contour lines with levels at the values specified in V.
Fills the len(V)-1 regions between the level values in sequence V.
Uses keyword arguments to control common line properties (colors, line width, origin, color map, and so on).

There exist certain constraints on the dimensionality and shape of X, Y, and Z. For example, X and Y can be of two dimensions and of the same shape as Z. If they are of one dimension, such that the length of X is equal to the number of columns in Z, then the length of Y will be equal to the number of rows in Z.

How to do it...

In the following code example, we will:

1. Implement a function to act as a mock signal processor.
2. Generate some linear signal data.
3. Transform the data into suitable matrices for use in matrix operations.
4. Plot contour lines.
5. Add contour line labels.
6. Show the plot.
7. Import numpy as np.
8. Import matplotlib as mpl.
9. Import matplotlib.pyplot as plt.

```
def process_signals(x,y):
return (1 – (x ** 2 + y ** 2)) * np.exp(-y ** 3 / 3)
x = np.arange(-1.5, 1.5, 0.1) y = np.arange(-1.5, 1.5, 0.1)
# Make grids of points X,Y = np.meshgrid(x, y)
Z = process_signals(X, Y)
# Number of isolines
N = np.arange(-1, 1.5, 0.3)

# adding the Contour lines with labels
CS = plt.contour(Z, N, linewidths=2, cmap=mpl.cm.jet) plt.clabel(CS,
inline=True, fmt='%1.1f', fontsize=10) plt.colorbar(CS)

plt.title('My function: $z=(1-x^2+y^2) e^{-(y^3)/3}$') plt.show()
```

This will give us the following chart:

How it works...

We reached for little helpers from numpy to create our ranges and matrices. After we evaluated my_function into Z, we simply called contour, providing Z and the number of levels for isolines.

At this point, try experimenting with the third parameter in the N arange() call. For example, instead of N = np.arange(-1, 1.5, 0.3), try changing 0.3 to0.1 or 1 to experience how the same data is seen differently, depending on how we encode the data in a contour plot.

We also added a color map by simply giving it CS (a matplotlib.contour. QuadContourSet instance).

Filling an under-plot area

The basic way to draw a filled polygon in matplotlib is to use matplotlib.pyplot.fill. This function accepts similar arguments as matplotlib.pyplot.plot—multiple x and y pairs and other Line2D properties. This

function returns the list of Patch instances that were added.

In this recipe we will learn how to shade certain areas of plot intersections.

Getting ready

matplotlib provides several functions to help us plot filled figures, apart from plotting functions that are inherently plotting closed filled polygons, such as histogram (), of course.

We already mentioned one— matplotlib.pyplot.fill—but there are the matplotlib.pyplot.fill_between() and matplotlib.pyploy.fill_betweenx() functions too. These functions fill the polygons between two curves. The main difference between fill_between() and fill_betweenx() is that the latter fills between the x axis values, whereas the former fills between the y axis values.

The function fill_between accepts argument x—an x axis array of data—and y1 and y2—the y axis arrays of the data. Using arguments, we can specify conditions under which the area will be filled. This condition is the Boolean condition, usually specifying the y axis value ranges. The default value is None—meaning, to fill everywhere.

How to do it...

To start off with a simple example, we will fill the area under a simple function:

```
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt

t = range(1000)
y = [sqrt(i) for i in t]
plt.plot(t, y, color='red', lw=2) plt.fill_between(t, y, color='silver') plt.show()
```

The previous code gives us the following plot:

This is fairly straightforward and gives an idea how fill_between() works. Note how we needed to plot the actual function line (using plot(), of course), where fill_between() just draws a polygonal area filled with color ('silver').
We will demonstrate another recipe here. It will involve more conditioning for the fill function. The following is the code for the example:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0.0, 2, 0.01) y1 = np.sin(np.pi*x)
y2 = 1.7*np.sin(4*np.pi*x)

fig = plt.figure()
axes1 = fig.add_subplot(211)
axes1.plot(x, y1, x, y2, color='grey')
axes1.fill_between(x, y1, y2, where=y2<=y1, facecolor='blue', interpolate=True)
axes1.fill_between(x, y1, y2, where=y2>=y1, facecolor='gold', interpolate=True)
```

```
axes1.set_title('Blue where y2 <= y1. Gold-color where y2 >= y1.')
axes1.set_ylim(-2,2)

# Mask values in y2 with value greater than 1.0
y2 = np.ma.masked_greater(y2, 1.0)
axes2 = fig.add_subplot(212, sharex=axes1)
axes2.plot(x, y1, x, y2, color='black')
axes2.fill_between(x, y1, y2, where=y2<=y1, facecolor='blue', interpolate=True)
axes2.fill_between(x, y1, y2, where=y2>=y1, facecolor='gold', interpolate=True)
axes2.set_title('Same as above, but mask')
axes2.set_ylim(-2,2)
axes2.grid('on')

plt.show()
```

The preceding code will render the following plot:

How it works...

For this example, we first created two sinusoidal functions that overlap at certain points. We also created two subplots to compare the two variations that render filled regions.

In both cases, we used fill_between() with an argument, where, that accepts an

N-length Boolean array and will fill over the regions where where equals True.

The bottom subplot illustrates mask_greater, which masks an array at values greater than a given value. This is a function from the numpy.ma package to handle missing or invalid values. We turned the grid on the bottom axes to make it easier to spot this.

Drawing polar plots

If the data is already represented using polar coordinates, we can as well display it using polar figures. Even if the data is not in polar coordinates, we should consider converting it to polar form and draw on polar plots.
To answer whether we want to do this, we need to understand what the data represents and what we are hoping to display to the end user. Imagining what the user will read and decode from our figures leads us usually to the best of visualizations.

Polar plots are commonly used to display information that is radial in nature. For example, in sun path diagrams—we see the sky in radial projection, and the radiation maps of antennas radiate differently at different angles. You can learn more about this at: http://www. astronwireless.com/topic-archives-antenna-radiation-patterns.asp.

In this recipe, we will learn how to change the coordinate system used in the plot and to use the polar coordinate system instead.

Getting ready

To display data in polar coordinates, we must have appropriate data values. In the polar coordinate system, a point is described with radius distance (usually denoted with *r*) and angle (usually *theta*). The angle can be in radians or degrees, but matplotlib uses degrees.

Similar enough to the function plot(), to draw polar plots, we will use the function polar(), which accepts two same-length arrays of parameters, theta and r, for angle array and radius array, respectively. The function also accepts other formatting arguments, the same ones as plot() one does.

We also need to tell matplotlib that we want axes in the polar coordinate system. This is done by providing the polar=True argument to the add_axes or

add_subplot functions.

Additionally, to set other properties on the figure, such as grids on radii or angles, we need to use matplotlib.pyplot.rgrids() to toggle radial grid visibility or to set up labels. Similarly, we use matplotlib.pyplot.thetagrid() to configure angle ticks and labels.

How to do it...

Here is one recipe that demonstrates how to plot polar bars:

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

figsize = 7
colormap = lambda r: cm.Set2(r / 20.) N = 18 # number of bars

fig = plt.figure(figsize=(figsize,figsize)) ax = fig.add_axes([0.2, 0.2, 0.7, 0.7],
polar=True)

theta = np.arange(0.0, 2*np.pi, 2*np.pi/N) radii = 20*np.random.rand(N)
width = np.pi/4*np.random.rand(N)
bars = ax.bar(theta, radii, width=width, bottom=0.0) for r, bar in zip(radii, bars):

bar.set_facecolor(colormap(r))
bar.set_alpha(0.6)
plt.show()
```

The preceding code snippet will give us the following plot:

How it works...

First, we create a square figure and add the polar axes to it. The figure does not have to be square, but then our polar plot will be ellipsoidal.

We then generate random values for a set of angles (theta) and a set of polar distances (radii). Because we drew bars, we also needed a set of widths for each bar, so we also generated a set of widths. Since maplotlib.axes.bar accepts an array of values (as almost all the drawing functions in matplotlib do) we don't have to loop over this generated dataset; we just need to call the bar once with all the arguments passed to it.

In order to make every bar easily distinguishable, we have to loop over each bar added to ax (Axes) and customize its appearance (face-color and transparency).

Visualizing the filesystem tree using a polar bar

We want to show in this recipe how to solve a "real-world" task—how to use matplotlib to visualize our directory occupancy.
In this recipe we will learn how to visualize a filesystem tree with relative sizes.

Getting ready

We all have big hard drives that sometimes contain stuff that we usually forget about. It would be nice to see what is inside such a directory, and what the biggest file inside is. Although there are many more sophisticated and elaborate software products for this job, we want to demonstrate how this is achievable using Python and matplotlib.

How to do it...

Let's perform the following steps:

1. Implement a few helper functions to deal with folder discovery and internal data structures.

2. Implement the main function, draw(), that does the plotting.

3. Implement the main program body that verifies the user input arguments:

import os
import sys

import matplotlib.pyplot as plt import matplotlib.cm as cm import numpy as np

def build_folders(start_path): folders = []

for each in get_directories(start_path):
size = get_size(each)
if size >= 25 * 1024 * 1024:

folders.append({'size' : size, 'path' : each})

for each in folders:
print "Path: " + os.path.basename(each['path']) print "Size: " + str(each['size'] / 1024 / 1024) + " MB" return folders

def get_size(path):
assert path is not None
total_size = 0
for dirpath, dirnames, filenames in os.walk(path): for f in filenames:

fp = os.path.join(dirpath, f)
try:
size = os.path.getsize(fp)
total_size += size
#print "Size of '{0}' is {1}".format(fp, size) except OSError as err:

```
print str(err)
pass
return total_size

def get_directories(path):
dirs = set()
for dirpath, dirnames, filenames in os.walk(path):

dirs = set([os.path.join(dirpath, x) for x in dirnames]) break # we just want the
first one
return dirs

def draw(folders):
""" Draw folder size for given folder"""
figsize = (8, 8) # keep the figure square
ldo, rup = 0.1, 0.8 # leftdown and right up normalized fig =
plt.figure(figsize=figsize)
ax = fig.add_axes([ldo, ldo, rup, rup], polar=True)

# transform data
x = [os.path.basename(x['path']) for x in folders] y = [y['size'] / 1024 / 1024 for y
in folders] theta = np.arange(0.0, 2 * np.pi, 2 * np.pi / len(x)) radii = y

bars = ax.bar(theta, radii)
middle = 90/len(x)
theta_ticks = [t*(180/np.pi)+middle for t in theta] lines, labels =
plt.thetagrids(theta_ticks, labels=x,

frac=0.5)
for step, each in enumerate(labels):
each.set_rotation(theta[step]*(180/np.pi)+ middle) each.set_fontsize(8)

# configure bars
colormap = lambda r:cm.Set2(r / len(x)) for r, each in zip(radii, bars):

each.set_facecolor(colormap(r)) each.set_alpha(0.5)
plt.show()
```

4. Next, we will implement the main program body where we verify the input arguments given by the user when the program is called from the command line:

```
if __name__ == '__main__':
if len(sys.argv) is not 2:
print "ERROR: Please supply path to folder."
sys.exit(-1)

start_path = sys.argv[1]

if not os.path.exists(start_path): print "ERROR: Path must exits." sys.exit(-1)

folders = build_folders(start_path)

if len(folders) < 1:
print "ERROR: Path does not contain any folders." sys.exit(-1)

draw(folders)
```

You need to run the following from the command line: **$ python ch04_rec11_filesystem.py /usr/lib/**

It will produce a plot similar to this one:

How it works...

We will start from the bottom of the code, after if __name__ == '__main__' because that is the place where our program starts.
Using the module sys, we pick up command-line arguments; they represent the path to directory we want to visualize.

The function build_folders builds the list of dictionaries, each containing the size and path that it found inside the given start_path. This function calls get_directories, which returns a list of all the subdirectories in start_path. Later, for each found directory, we calculated the sizes in bytes using the get_size function.

For debugging purposes, we print our dictionary so that we are able to compare the figure to what our data looks like.

After we have built the folders as a list of dictionaries, we pass them to a function, draw, that performs all the work of transforming the data to the right dimensions (here, we are using the polar coordinate system), constructing the polar figure, and drawing all the bars, ticks, and labels.

Strictly speaking, we should divide this job into smaller functions, especially if this code is to be further developed.

5

Making 3D Visualizations

We will learn the following recipes in this chapter:
f Creating 3D bars

f Creating 3D histograms
f Animating in matplotlib
f Animating with OpenGL

Introduction

Visualization in 3D is sometimes effective, and sometimes inevitable. Here, we present some examples that will satisfy the most frequent requirements.
The content of this chapter will introduce and explain some topics on 3D visualizations.

Creating 3D bars

Although matplotlib is mainly focused on plotting and mainly in two dimensions, there are different extensions that enable us to plot over geographical maps, to integrate more with Excel and plot in 3D. These extensions are called toolkits in the matplotlib world. Toolkit is a collection of specific function that focuses on one topic, such as plotting in 3D.

Popular toolkits are Basemap, GTK Tools, Excel Tools, Natgrid, AxesGrid, and mplot3d. We will explore more of mplot3d in this recipe. The toolkit mpl_toolkits.mplot3d provides some basic 3D plotting. Plots supported are scatter, surf, line, and mesh. Although this is not the best 3D plotting library, it comes with matplotlib and we are already familiar with the interface.

Getting ready

Basically, we still need to create a figure and add desired axes to it. The difference is that we specify 3D projection for the figure, and the axes we add are Axes3D.

Now, we can use almost the same functions for plotting. Of course, what are different are the arguments, for we now have three axes that we need to provide data for.

For example, the function mpl_toolkits.mplot3d.Axes3D.plot specifies xs, ys, zs, and zdir arguments. All others are transferred directly to matplotlib.axes.Axes.plot. We will explain these specific arguments:

f xs and ys: These are coordinates for x and y axes
f zs: This is the value(s) for z axis. It can be one for all points or one for each point

f zdir: This will choose what will be the z-axis dimension (usually this is zs, but can be xs or ys)

There is a method rotate_axes in the module mpl_toolkits. mplot3d.art3d that contains 3D artist code and functions to convert 2D artists into 3D versions, which can be added to Axes3D to reorder coordinates so that the axes are rotated along with zdir. The default value is z. Prepending the axis with a '-' does the inverse transform, so zdir can be x, -x, y, -y, z, or -z.

How to do it...

This is the code to demonstrate the concept explained:

```
import random

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt import matplotlib.dates as mdates

from mpl_toolkits.mplot3d import Axes3D

mpl.rcParams['font.size'] = 10
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

for z in [2011, 2012, 2013, 2014]: xs = xrange(1,13)
ys = 1000 * np.random.rand(12)

color = plt.cm.Set2(random.choice(xrange(plt.cm.Set2.N))) ax.bar(xs, ys, zs=z,
zdir='y', color=color, alpha=0.8)
```

```
ax.xaxis.set_major_locator(mpl.ticker.FixedLocator(xs))
ax.yaxis.set_major_locator(mpl.ticker.FixedLocator(ys))

ax.set_xlabel('Month')
ax.set_ylabel('Year')
ax.set_zlabel('Sales Net [usd]')

plt.show()
```

The preceding code produces the following diagram:

How it works...

We had to do the same preparation work as in the 2D world. The difference here is that we needed to specify the kind of backend. Then we generate some random data, for example, 4 years of sale (2011-2014).

We needed to specify the Z values to be the same for the 3D axis.
We picked a color randomly from the color-map set, and then we associated each Z-order collection of xs, ys pairs we would render the bar series.

There's more...

Other plotting from 2D matplotlib is available here; for example, scatter() with similar interface to plot() but with added size of the point marker. We are also familiar with contour, contourf, and bar.

The new types that are available just in 3D are wireframe, surface, and tri-surface plots. The following code example plots tri-surface plot of popular Pringle functions or, more mathematically, hyperbolic paraboloid:

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.pyplot as plt
import numpy as np

n_angles = 36 n_radii = 8

# An array of radii
# Does not include radius r=0, this is to eliminate duplicate points radii =
np.linspace(0.125, 1.0, n_radii)

# An array of angles
angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
# Repeat all angles for each radius
angles = np.repeat(angles[...,np.newaxis], n_radii, axis=1)

# Convert polar (radii, angles) coords to cartesian (x, y) coords # (0, 0) is added
here. There are no duplicate points in the (x, y) plane
x = np.append(0, (radii*np.cos(angles)).flatten())
y = np.append(0, (radii*np.sin(angles)).flatten())
# Pringle surface

z = np.sin(-x*y)
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_trisurf(x, y, z, cmap=cm.jet, linewidth=0.2)
plt.show()
```

The preceding code will give the following output:

Creating 3D histograms

Similar to 3D bars, we might want to create 3D histograms. They are used to easily spot correlation between three independent variables. They can be used to extract information from images where the third dimension could be the intensity of a channel in (x,y) space of the image under analysis.

In this recipe, we will learn how to create 3D histograms.

Getting ready

To recall, a histogram represents a number of occurrences of some value in a particular column (usually called "bin"). The three-dimensional histogram, then, represents a number of occurrences in a grid. This grid is rectangular, over two variables that are data in the two columns.

How to do it...

For this computation, we will:
1. Use NumPy because it has the function to compute a histogram of two variables.

2. Generate x and y from normal distributions but with different parameters to be able to distinct the correlation in the resulting histogram.
3. Plot the scatter plot of the same dataset to demonstrate how different is the display of scatter plot to 3D histogram.
Following is the code sample to implement the described steps:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

from mpl_toolkits.mplot3d import Axes3D
mpl.rcParams['font.size'] = 10
samples = 25
x = np.random.normal(5, 1, samples) y = np.random.normal(3, .5, samples)
fig = plt.figure()
ax1 = fig.add_subplot(211, projection='3d')
# compute two-dimensional histogram
hist, xedges, yedges = np.histogram2d(x, y, bins=10)

# compute location of the x,y bar positions
elements = (len(xedges) - 1) * (len(yedges) - 1) xpos, ypos =
np.meshgrid(xedges[:-1]+.25, yedges[:-1]+.25)

xpos = xpos.flatten() ypos = ypos.flatten() zpos = np.zeros(elements)

# make every bar the same width in base dx = .1 * np.ones_like(zpos)
dy = dx.copy()

# this defines the height of the bar dz = hist.flatten()

ax1.bar3d(xpos, ypos, zpos, dx, dy, dz, color='b', alpha=0.4) ax1.set_xlabel('X
Axis')
ax1.set_ylabel('Y Axis')
ax1.set_zlabel('Z Axis')

# plot the same x,y correlation in scatter plot # for comparison
```

```
ax2 = fig.add_subplot(212)
ax2.scatter(x, y)
ax2.set_xlabel('X Axis')
ax2.set_ylabel('Y Axis')

plt.show()
```

The preceding code will give the following output:

How it works...

We prepare a computer histogram using np.histogram2d that returns our histogram (hist) and the x and y bin edges.

Because for the bard3d function we need coordinates in x, y space, so we need to compute the common matrix coordinates, and for that we use np.meshgrid that combines the x and y positional vectors into the 2D space grid (matrix). We can use this to plot bars in the xy plane locations.

The variables dx and dy represent width of the base of each bar and we want to make this constant, hence we give it a 0.1 point value for every position in the xy plane. The value in the z axis (dz) is actually our computer histogram (in variable hist) that represents the count of common x and y samples at a particular bin.
Scatter plot below (in preceding plots) displays the 2D axes that also visualize

correlation between two similar distributions but with a different set of starting parameters.

Sometimes 3D is what gives us more information and resonates what the data is containing in a better way. As more often 3D visualizations are more confusing than 2D, it is advised to think twice before we choose them over 2D.

Animating in matplotlib

In this recipe, we will explore how to animate our figures. Sometimes it is more descriptive to have pictures moving in animations to explain what is going on when we change values of variables. Our main library has limited but usually sufficient animation capabilities and we will explain how to use them.

Getting ready

A framework for animation is added to the standard matplotlib from version 1.1 and its main class is matplotlib.animation.Animation. This class is a base class; it is to be subclassed for specific behavior as is the case with the already provided classes: TimedAnimation, ArtistAnimation, and FuncAnimation. The following table gives the description of the classes:

Class name (parent class) Animation (object)

Description
This class wraps the creation of an animation using matplotlib. It is only a base class that should be subclassed to provide the needed behavior.
Class name (parent class)
TimedAnimation (Animation)

ArtistAnimation (TimedAnimation)
FuncAnimation (TimedAnimation)

Description
This animation subclass supports time-based animation, and drawing a new frame every interval* milliseconds.
Before calling this function, all plotting should have taken place and the relevant artists saved.
This makes an animation by repeatedly calling a function, passing in (optional)

arguments.

In order to be able to save animations in a video file, we must have ffmpeg or mencoder installer. Installation of these packages varies depending on the OS used, and the changes by different releases, so we must leave it to our dear reader to Google valid information.

How to do it...

The following code listing demonstrates some matplotlib animations:

```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

fig = plt.figure()
ax = plt.axes(xlim=(0, 2), ylim=(-2, 2)) line, = ax.plot([], [], lw=2)

def init():
"""Clears current frame."""" line.set_data([], []) return line,

def animate(i):
"""Draw figure.
@param i: Frame counter
@type i: int
"""
x = np.linspace(0, 2, 1000)
y = np.sin(2 * np.pi * (x - 0.01 * i)) * np.cos(22 * np.pi * (x -

0.01 * i))
line.set_data(x, y)
return line,
# This call puts the work in motion
# connecting init and animate functions and figure we want to draw animator =
animation.FuncAnimation(fig, animate, init_func=init,

frames=200, interval=20, blit=True)

# This call creates the video file.
# Temporary, every frame is saved as PNG file # and later processed by ffmpeg
encoder into MPEG4 file # we can pass various arguments to ffmpeg via
```

```
extra_args animator.save('basic_animation.mp4', fps=30,

extra_args=['-vcodec', 'libx264'], writer='ffmpeg_file')
plt.show()
```

This will create the file basic_animation.mp4 in the folder from where you started working on this file, and also display a figure window with a running animation. The video file can be opened with most modern video players that support MPEG-4 format. The figure (frame) should look like the following graph:

How it works...

Most important are the functions init(), animate(), and save(). We first construct FuncAnimate by passing two callback functions to it, init and animate. Then we call its save() method to save our video file. More details on each function are in the following table:

Function name init

animate
matplotlib.animation.Animation. save

Usage
Passed to the matplotlib.animation. FuncAnimation constructor via parameter init_func to clear frame before next frame is drawn.
Passed to the matplotlib.animation. FuncAnimation constructor via func parameter.
The figure we want to animate is passed

via fig argument, which under the hood is passed to the matplotlib.animation. Animation constructor to connect
animation events with the figure we want to draw. This function gets (optional) parameters from frames, usually iterable, representing a number of frames.

Saves a movie file by drawing every frame. It creates temporary image files before processing them through encoder (ffmpeg or mencoder) to create a video file. This function also accepts various parameters that configure the video output, metadata (author...), codec to use, resolution/size, and so on. One of the parameters is the one that defines what video encoder to use. Currently supported are ffmpeg, ffmpeg_file, and mencoder.

There's more...

The usage of matplotlib.animation.ArtistAnimation differs from the usage of FuncAnimation, in that we must draw each artist beforehand and then instantiate the ArtistAnimation class with all artists' different frames. Artist animation is a kind of a wrapper of the matplotlib.animation.TimedAnimation class that draws frames every N milliseconds, thus supporting time-based animation. Unfortunately, for the Mac OS X users, animation framework can be troublesome on this platform, and sometimes simply does not work, which will improve with future releases of matplotlib.

Animating with OpenGL

The motivation to use OpenGL comes from the limitations of CPU processing power when we face a task to visualize millions of data points and do it fast (sometimes even in real time).

Modern computers have powerful GPUs that are made for fast visualization related computations (such as games), and there is no reason why they can't be used for scientific-related visualizations.

Actually, there is at least one drawback of writing the hardware accelerated software. As far as hardware dependency is concerned, modern graphic cards require proprietary drivers, sometimes not available on the target platform/machine (for example, user laptop); even when available, sometimes installing required dependencies on site is not what you want to spend your time on, while all you want is to present your findings and demonstrate your research results. This is not a show stopper but have this in mind, and measure benefits and costs of introducing this complexity in your project.

With caveats explained, we can say yes to hardware accelerated visualizations and say yes to OpenGL, industry standard for accelerated graphics.

We will be using OpenGL, as it is across platform, so the examples should work as presented on Linux, Mac, or Windows, given that you have the required hardware and OS level drivers installed.

Getting ready

If you have never used OpenGL, we will now try to give a quick intro, although really to know OpenGL, at least one full book needs to be read and understood. OpenGL is a specification, not an implementation, so OpenGL itself doesn't have any code, where the implementations are libraries developed according to this specification. Those are shipped with your operating system, or by vendors of graphic cards such as NVIDIA, or AMD/ATI.

Moreover, OpenGL is concerned only with graphics rendering and not animation, timing and other complex things that are left for additional library to pick up.

Basics of animating with OpenGL

Because OpenGL is a rendering library, it does not know what objects we draw on a screen. It doesn't care if we draw a cat, a ball, or a line, or all of those objects. So to move a rendered object, we need to clear and draw the whole image again. To animate something, we need a loop that draws and redraws everything very fast, and displays that to a user, so that the user thinks he/ she is seeing an animation.

Installing OpenGL on a machine is a platform dependent process. On Mac OS X, OpenGL implementations is part of OS upgrade, but development libraries (so called "headers") are part of Xcode development package.

On Windows, the best way would be to install vendor's latest graphic drivers for your graphic card. OpenGL may work without them, but you will probably be left without the latest features of stock drivers.

On Linux, if you are not against installing closed source software, there are vendor specific drivers downloadable from either Distro's own software manager, or from vendor site as installable binary. Standard implementation is almost always Mesa3D, the best known OpenGL implementation that uses Xorg to provide support for OpenGL for Linux, FreeBSD, and similar operating systems.

Basically, on Debian/Ubuntu, you should install the following packages and their dependencies: **$ sudo apt-get install libgl1-mesa-dev libgl-mesa-dri**

After this, you should be ready to use some development libraries and/or frameworks to actually write OpenGL backed applications.

We are focused here on Python, so we will overview some of the Python's most used libraries and frameworks that are built on top of OpenGL. We will mention matplotlib and its current and future support for OpenGL:

f Mayavi: This is a library specialized for 3D
f Pyglet: This is a pure Python library for graphics
f Glumpy: This is a fast rendering library built on top of NumPy
f Pyglet and OpenGL: This is used for visualizing Big Data (multi-million data points)

How to do it...

The specialized project Mayavi is a full feature 3D graphic library, and it is mainly used for advanced 3D rendering. It comes with the already mentioned Python packages such as EPD (though not with free license), which is a recommended way to install it on Windows and Mac OS X. On Linux; it can also be easily installed using pip:

$ pip install mayavi

Mayavi can be used as a development library/framework or as an application. The Mayavi application comprises a visual editor for easy data explorations and somewhat interactive visualization.

As a library it can be used similarly to matplotlib, either from a script interface or as a full object-oriented library. Most of that interface is inside the module mlab, to be able to use that interface. For example, a simple animation with Mayavi can be done as follows:

```
import numpy
from mayavi.mlab import *

# Produce some nice data.
n_mer, n_long = 6, 11
pi = numpy.pi
dphi = pi/1000.0
phi = numpy.arange(0.0, 2*pi + 0.5*dphi, dphi, 'd') mu = phi*n_mer
x = numpy.cos(mu)*(1+numpy.cos(n_long*mu/n_mer)*0.5) y =
numpy.sin(mu)*(1+numpy.cos(n_long*mu/n_mer)*0.5) z =
numpy.sin(n_long*mu/n_mer)*0.5

# View it.
l = plot3d(x, y, z, numpy.sin(mu), tube_radius=0.025, colormap='Spectral')

# Now animate the data.
ms = l.mlab_source
for i in range(100):

x = numpy.cos(mu)*(1+numpy.cos(n_long*mu/n_mer + numpy.pi*(i+1)/5.)*0.5)
scalars = numpy.sin(mu + numpy.pi*(i+1)/5)
ms.set(x=x, scalars=scalars)
```

The preceding code will produce the following window with the rotating figure:

How it works...

We generate the dataset and create a set of functions for x, y, and z to be used in the plot3d function for the start position of the figure.

We then import the mlab_source object that enables us to manipulate our plot on the level of the points and scalars. We use this feature to then set particular points and scalars in for the loop to create a rotation animation with 100 frames.

There's more...

If you want to experiment more, the easiest way to do that is to load IPython, import mayavi.mlab, and run some test_* functions.
To see what is going on, you use IPython's ability to inspect and explore Python source, as shown in the following code:

```
In [1]: import mayavi.mlab

In [2]: mayavi.mlab.test_simple_surf??
Type: function
String Form:<function test_simple_surf at 0x641b410>
File: /usr/lib/python2.7/dist-packages/mayavi/tools/helper_ functions.py
Definition: mayavi.mlab.test_simple_surf()
Source:
def test_simple_surf():

"""Test Surf with a simple collection of points."""  x, y =
numpy.mgrid[0:3:1,0:3:1]
```

```
return surf(x, y, numpy.asarray(x, 'd'))
```

We see here how by adding two question marks after function name ("??"), IPython found the source of the function and showed it to us. This is a true exploratory computing, and is often used within the visualization community because it is a fast way to get to know your data and code.

Using Pyglet Quickstart

Pyglet is another popular Python library that eases writing graphics and windowing-related applications. It supports OpenGL through its module pyglet.gl, but you don't have to directly talk to this module to use the power of Pyglet. Most convenient use is through pyglet.graphics.

Pyglet takes a different approach to Mayavi; there is no visual IDE, and you are responsible for doing everything from creating a window to issuing a low-level OpenGL call to configure OpenGL context. This is sometimes slower than Mayavi, but what you gain is the ability to control every piece of your application. Sometimes that also means more work hours involved, but usually it means better quality and performance for your applications.

The simplest application (image viewer) can be obtained using the following code:

```
import pyglet
window = pyglet.window.Window()
image = pyglet.resource.image('kitten.jpg')

@window.event
def on_draw():
window.clear()
image.blit(0, 0)

pyglet.app.run()
```

Here you can see that we create a window, load an image, and define what is going to happen when we draw a window object (that is, we define an event handler for the on_draw event). Lastly, we run our application (pyglet.app.run()).

Under the hood, OpenGL is used to draw in windows. This interface is accessible from the pyglet.gl module. Using it directly, though, is not efficient, so pyglet has a simpler interface at pyglet.graphics, where vertex arrays and

buffers are used internally to this interface.

Using Glumpy Quickstart
Glumpy is an OpenGL plus NumPy library for fast NumPy visualization using OpenGL. It is an open source project started by *Nicolas Rougier*, and aimed to be efficient. To use it, we need Python OpenGL bindings, SciPy, and of course Glumpy. Use following commands for this:

sudo apt-get install python-opengl
sudo pip install scipy
sudo pip install glumpy

Glumpy uses OpenGL textures to represent arrays since it is probably the fastest method of visualization on modern graphic hardware.
Pyprocessing introduction

Pyprocessing works in much the same fashion as Processing (http://processing.org). Most functions in Pyprocessing are equivalent to Processing functions. If you are familiar with Processing and Python, you already know almost everything you need to write Pyprocessing applications. To use it, the only thing we need to do is to import the pyprocessing package, write the rest of your code using Pyprocessing functions and data structures, and call run().

There are a lot of free tutorials on OpenGL and how to use it from C/C++ or any other language bindings. One list is provided here, on official OpenGL wiki at http://www. opengl.org/wiki/Getting_started#Tutorials_and_How_To_Guides.

There are many more projects that tackle Python, OpenGL, and 3D visualization in general. Some of them are young, some of them not maintained, but if you spot one that needs to be mentioned, let us know.

6

Plotting Charts with Images and Maps

This chapter contains recipes that will show:
f Processing images with PIL

f Plotting with images
f Displaying image with other plots in the figure
f Plotting data on a map using Basemap
f Plotting data on a map using Google Map API
f Generating CAPTCHA images

Introduction

This chapter explores how to work with images and maps. Python has some well-known image libraries that allow us to process images in both aesthetic and scientific ways. We will touch on PIL's capabilities by demonstrating how to process images by applying filters and by resizing them.
Furthermore, we will show how to use image files as annotation for our matplotlibs' charts. To deal with data visualization of geospatial datasets, we will cover the functionality of Python's available libraries and public APIs that we can use with map-based visual representations. The final recipe shows how Python can create CAPTCHA test images.

Processing images with PIL

Why use Python for image processing, if we could use WIMP (http://en.wikipedia.org/ wiki/WIMP_(computing)) or WYSIWYG (http://en.wikipedia.org/wiki/WYSIWYG) to achieve the same goal? This is used because we want to create an automated system to process images in real time without human support, thus, optimizing the image pipeline.

Getting ready

Note that the PIL coordinate system assumes that the (0,0) coordinate is in the upper-left corner.

The Image module has a useful class and instance methods to perform basic operations over a loaded image object (im):
f im = Image.open(filename): This opens a file and loads the image into im object.

f im.crop(box): This crops the image inside the coordinates defined by box. box defines left, upper, right, lower pixels coordinates (for example: box = (0, 100, 100,100)).

f im.filter(filter): This applies a filter on the image and returns a filtered image.

f im.histogram(): This returns a histogram list for this image, where each item represents the number of pixels. Number of items in the list is 256 for single channel images, but if the image is not a single channel image, there can be more items in the list. For an RGB image the list contains 768 items (one set of 256 values for each channel).

f im.resize(size, filter): This resizes the image and uses a filter for resampling. The possible filters are NEAREST, BILINEAR, BICUBIC, and ANTIALIAS. The default is NEAREST.

f im.rotate(angle, filter): This rotates an image in the counter clockwise direction.
f im.split(): This splits bands of image and returns a tuple of individual bands. Useful for splitting an RGB image into three single band images.

f im.transform(size, method, data, filter): This applies transformation on a given image using data and a filter. Transformation can be AFFINE, EXTENT, QUAD, and MESH. You can read more about transformation in the official documentation. Data defines the box in the original image where the transformation will be applied.

The ImageDraw module allows us to draw over the image, where we can use functions such as arc, ellipse, line, pieslice, point, and polygon to modify the pixels of the loaded image. The ImageChops module contains a number of image channel operations (hence the name Chops) that can be used for image composition, painting, special effects, and other processing operations. Channel operations are allowed only for 8-bit images. Here are some interesting channel operations:

f ImageChops.duplicate(image): This copies current image into a new i mage object
f ImageChops.invert(image): This inverts an image and returns a copy f ImageChops.difference(image1, image2): This is useful for verification that images are the same without visual inspection

The ImageFilter module contains the implementation of the kernel class that allows the creation of custom convolution kernels. This module also contains a set of healthy common filters that allows the application of well-known filters (BLUR and MedianFilter) to our image.

There are two types of filters provided by the ImageFilter module: fixed image enhancement filters and image filters that require certain arguments to be defined; for example, size of the kernel to be used.

We can easily get the list of all fixed filter names in IPython:
In [1]: import ImageFilter
In [2]: [f for f in dir(ImageFilter) if f.isupper()] Out[2]:
['BLUR',

'CONTOUR',
'DETAIL',
'EDGE_ENHANCE',
'EDGE_ENHANCE_MORE',
'EMBOSS',
'FIND_EDGES',
'SHARPEN',
'SMOOTH',
'SMOOTH_MORE']

The next example shows how we can apply all currently supported fixed filters on any supported image:

```
import os
import sys
from PIL import Image, ImageChops, ImageFilter

class DemoPIL(object):
def __init__(self, image_file=None): self.fixed_filters = [ff for ff in
dir(ImageFilter) if ff.isupper()]
```

```
assert image_file is not None
assert os.path.isfile(image_file) is True self.image_file = image_file
self.image = Image.open(self.image_file)

def _make_temp_dir(self):
from tempfile import mkdtemp
self.ff_tempdir = mkdtemp(prefix="ff_demo")

def _get_temp_name(self, filter_name):
name, ext = os.path.splitext(os.path.basename(self.image_ file))

newimage_file = name + "-" + filter_name + ext path =
os.path.join(self.ff_tempdir, newimage_file) return path

def _get_filter(self, filter_name):

# note the use Python's eval() builtin here to return function object
real_filter = eval("ImageFilter." + filter_name)
return real_filter

def apply_filter(self, filter_name):
print "Applying filter: " + filter_name filter_callable =
self._get_filter(filter_name) # prevent calling non-fixed filters for now if
filter_name in self.fixed_filters:

temp_img = self.image.filter(filter_callable) else:
print "Can't apply non-fixed filter now." return temp_img

def run_fixed_filters_demo(self):
self._make_temp_dir()
for ffilter in self.fixed_filters:

temp_img = self.apply_filter(ffilter)
temp_img.save(self._get_temp_name(ffilter)) print "Images are in:
{0}".format((self.ff_tempdir),)

if __name__ == "__main__": assert len(sys.argv) == 2 demo_image =
sys.argv[1]
demo = DemoPIL(demo_image)
# will create set of images in temporary folder demo.run_fixed_filters_demo()
```

We can run this easily from the command prompt:
$ pythonch06_rec01_01_pil_demo.py image.jpeg

We packed our little demo in the DemoPIL class, so we can extend it easily while sharing the common code around the demo function run_fixed_filters_demo. Common code here includes opening the image file, testing if the file is really a file, creating temporary directory to hold our filtered images, building the filtered image filename, and printing useful information to user. This way the code is organized in a better manner and we can easily focus on our demo function, without touching other parts of the code.

This demo will open our image file and apply every fixed filter available in ImageFilter to it and save that new filtered image in a unique temporary directory. The location of this temporary directory is retrieved, so we can open it with our OS's file explorer and view the created images.

As an optional exercise, try extending this demo class to perform other filters available in ImageFilter on the given image.
How to do it...

The example in this section shows how we can process all the images in a certain folder. We specify a target path, and the program reads all the image files in that target path (images folder) and resizes them to a specified ratio (0.1 in this example), and saves each one in a target folder called thumbnail_folder:

```
import os
import sys
from PIL import Image

class Thumbnailer(object):
def __init__(self, src_folder=None): self.src_folder = src_folder

self.ratio = .3
self.thumbnail_folder = "thumbnails"
def _create_thumbnails_folder(self):
thumb_path = os.path.join(self.src_folder, self.thumbnail_ folder)
if not os.path.isdir(thumb_path): os.makedirs(thumb_path)

def _build_thumb_path(self, image_path):
root = os.path.dirname(image_path)
```

```
name, ext = os.path.splitext(os.path.basename(image_path)) suffix =
".thumbnail"
return os.path.join(root, self.thumbnail_folder, name + suffix

+ ext)

def _load_files(self):
files = set()
for each in os.listdir(self.src_folder):

each = os.path.abspath(self.src_folder + '/' + each) if os.path.isfile(each):
files.add(each)
return files
def _thumb_size(self, size):
return (int(size[0] * self.ratio), int(size[1] * self.ratio))

def create_thumbnails(self):
self._create_thumbnails_folder() files = self._load_files()

for each in files:
print "Processing: " + each
try:

img = Image.open(each)
thumb_size = self._thumb_size(img.size) resized = img.resize(thumb_size,
Image.ANTIALIAS) savepath = self._build_thumb_path(each)
resized.save(savepath)

except IOError as ex:
print "Error: " + str(ex)

if __name__ == "__main__":
# Usage:
# ch06_rec01_02_pil_thumbnails.py my_images assert len(sys.argv) == 2
src_folder = sys.argv[1]

if not os.path.isdir(src_folder):
print "Error: Path '{0}' does not exits.".format((src_folder)) sys.exit(-1)
thumbs = Thumbnailer(src_folder)
```

```
# optionally set the name of theachumbnail folder relative to *src_folder*.
thumbs.thumbnail_folder = "THUMBS"

# define ratio to resize image to
# 0.1 means the original image will be resized to 10% of its size thumbs.ratio =
0.1

# will create set of images in temporary folder thumbs.create_thumbnails()
```

How it works...

For the given src_folder folder, we load all the files in this folder and try to load each file using Image.open(); this is the logic of the create_thumbnails() function. If the file we try to load is not an image, IOError will be thrown, and it will print this error and skip to next file in the sequence.

If we want to have more control over what files we load, we should change the _load_ files() function to only include files with certain extension (file type):

```
for each in os.listdir(self.src_folder):
if os.path.isfile(each) and os.path.splitext(each) is in
('.jpg','.png'):
self._files.add(each)
```

This is not foolproof, as file extension does not define file type, it just helps the operating system to attach a default program to the file, but it works in majority of the cases and is simpler than reading a file header to determine the file content (which still does not guarantee that the file really is the first couple of bytes, say it is).

There's more...

With PIL, although not used very often, we can easily convert images from one format to the other. This is achievable with two simple operations: first open an image in a source format using open(), and then save that image in the other format using save(). Format is defined either implicitly via filename extension (.png or .jpeg), or explicitly via the format of the argument passed to the save() function.

Plotting with images

Images can be used to highlight the strengths of your visualization in addition to pure data values. Many examples have proven that by using symbolic images, we map deeper into the viewer's mental model, thereby helping the viewer to remember the visualizations better and for a longer time. One way to do so is to place images where your data is, to map the values to what they represent. The matplotlib library is capable of delivering this functionality, thus we will demonstrate how to do it.

Getting ready

Use the fictional example from the story *The Gospel of the Flying Spaghetti Monster*, by *Bobby Henderson* where the author correlates number of pirates with sea-surface temperature. To highlight this correlation, we will display the size of the pirate ship proportional to the value representing the number of pirates in the year the sea-surface temperature is measured.

We will use Python matplotlib library's ability to annotate using images and text with advanced location settings, as well as arrow capabilities.
All the files required in the following recipe are available in the source code repository in the ch06 folder.

How to do it...

The following example shows how to add an annotation to a chart using images and text:

```
import matplotlib.pyplot as plt
from matplotlib._png import read_png
from matplotlib.offsetbox import TextArea, OffsetImage, \

AnnotationBbox

def load_data():
import csv
with open('pirates_temperature.csv', 'r') as f:

reader = csv.reader(f)
header = reader.next()
datarows = []
for row in reader:

datarows.append(row)
```

```
return header, datarows

def format_data(datarows): years, temps, pirates = [], [], [] for each in datarows:

years.append(each[0])
temps.append(each[1])
pirates.append(each[2])

return years, temps, pirates
```

After we have defined helper functions, we can approach the construction of the figure object and add subplots. We will annotate these for every year in the collection of years using the image of the ship, scaling the image to the appropriate size:

```
if __name__ == "__main__":
fig = plt.figure(figsize=(16,8))
ax = plt.subplot(111) # add sub-plot

header, datarows = load_data()
xlabel, ylabel = header[0], header[1]
years, temperature, pirates = format_data(datarows) title = "Global Average
Temperature vs. Number of Pirates"

plt.plot(years, temperature, lw=2) plt.xlabel(xlabel)
plt.ylabel(ylabel)

# for every data point annotate with image and number for x in
xrange(len(years)):
# current data coordinate xy = years[x], temperature[x]
# add image
ax.plot(xy[0], xy[1], "ok")
# load pirate image
pirate = read_png('tall-ship.png')
# zoom coefficient (move image with size) zoomc = int(pirates[x]) * (1 / 90000.)
# create OffsetImage
imagebox = OffsetImage(pirate, zoom=zoomc)

# create anotation bbox with image and setup properties ab =
AnnotationBbox(imagebox, xy,
```

```
xybox=(-200.*zoomc, 200.*zoomc),
xycoords='data',
boxcoords="offset points",
pad=0.1,
arrowprops=dict(arrowstyle="->",
connectionstyle="angle,angleA=0,angleB=30,rad=3")
) ax.add_artist(ab)

# add text
no_pirates = TextArea(pirates[x], minimumdescent=False) ab =
AnnotationBbox(no_pirates, xy,

xybox=(50., -25.),
xycoords='data',
boxcoords="offset points",
pad=0.3,
arrowprops=dict(arrowstyle="->",

connectionstyle="angle,angleA=0,angleB=30,rad=3")
) ax.add_artist(ab)

plt.grid(1)
plt.xlim(1800, 2020) plt.ylim(14, 16)
plt.title(title)

plt.show()
```

The preceding code should give the following plot:

How it works...

We start by creating a figure of a decent size, that is, 16 x 8. We need this size to fit the images we want to display. Now, we load our data from the file, using the csv module. Instantiating the csv reader object, we can iterate over the data from the file row by row. Note how the first row is special, it is the header describing our columns. As we have plotted years on the x axis and temperature on the y axis, we read that:

xlabel, ylabel, _ = header
And use the following lines:
plt.xlabel(xlabel)
plt.ylabel(ylabel)

We used neat Python convention here to unpack the header into three variables, where by using _ for variable name, we indicate that we are not interested in the value of that variable.

We return the header and datarows lists from the load_data function to the main caller. Using the format_data() function, we read every item in the list, and add each separate entity (year, temperature, and number of pirates) into the relevant ID list for that entity. Year is displayed along the x axis, while temperature is on the y axis. The number of pirates is displayed as an image of a pirate ship, and

also to add precision, the value will be displayed. We plot year/temperature values using the standard plot() function, not adding anything more, apart from making the line a bit wider (2 pt).

We proceed then to add one image for every measurement and to illustrate the number of pirates for a given year. For this we loop over the range of values of length (range(len(years))) , plotting one black point on each year/temperature coordinate:

```
ax.plot(xy[0], xy[1], "ok")
```

The image of the ship is loaded from file into a suitable array format using the read_png helper function:

```
pirate = read_png('tall-ship.png')
```

We then compute zoom coefficient (zoomc) to enable us to scale the size of the image in proportion to the number of pirates for the current (pirates[x]) measurement. We also use the same coefficient to position the image along the plot.

The actual image is then instantiated inside OffsetImage—the image container with relative position to its parent (AnnotationBbox). AnnotationBbox is an annotation-like class, but instead of displaying just text as with the Axes.annotate function, it can display other OffsetBox instances. This allows us to load an image or text object in an annotation and locate it at a particular distance from the data point, as well as to use the arrowing capabilities (arrowprops) to precisely point to an annotated data point.

We supply the AnnotateBbox constructor with certain arguments: f Imagebox: This must be an instance of OffsetBox (for example, OffsetImage); it is the content of the annotation box

f xy: This is the data point coordinate that the annotation relates to
f xybox: This defines the location of the annotation box

f xycoords: This defines what coordinating system is used by xy (for example, data coordinates)
f boxcoords: This defines what coordinating system is used by xybox (for example, offset from the xy location)
f pad: This specifies the amount of padding
f arrowprops: This is the dictionary of properties for drawing an arrow

connection from an annotation-bounding box to a data point

We add text annotation to this plot, using the same data items from the pirates list, with a slightly different relative position. Most of the arguments of the second AnnotationBbox are the same—we adjust xybox and pad to locate the text to the opposite side of the line. The text is inside the TextArea class instance, this is similar to what we do with image, but with texttime.TextArea and OffsetImage inherit from the same parent class, OffsetBox.

We set the text in this TextArea instance tono_pirates and put it in our AnnotationBbox.

Displaying an image with other plots in the figure

This recipe will show how we can make simple yet effective usage of Python matplotlib library to process image channels and display per-channel histogram of an external image.

Getting ready

We have provided some sample images, but the code is ready to load any image file, provided it is supported by matplotlib's imread function.

In this recipe, we will learn how to combine different matplotlib plots to achieve functionality of a simple image viewer that displays an image histogram for red, green, and blue channels.

How to do it...

To show how to build an image histogram viewer, we are going to implement a simple class named ImageViewer and that class will contain helper methods to:

1. Load image.
2. Separate RGB channels from image matrix.
3. Configure figure and axes (subplots).
4. Plot channel histograms.
5. Plot the image.

The following code shows how to build an image histogram viewer:

```
import matplotlib.pyplot as plt
import matplotlib.image as mplimage
import matplotlib as mpl
import os
```

```
class ImageViewer(object): def __init__(self, imfile): self._load_image(imfile)
self._configure()

self.figure = plt.gcf()
t = "Image: {0}".format(os.path.basename(imfile)) self.figure.suptitle(t,
fontsize=20)

self.shape = (3, 2)

def _configure(self):
mpl.rcParams['font.size'] = 10
mpl.rcParams['figure.autolayout'] = False mpl.rcParams['figure.figsize'] = (9, 6)
mpl.rcParams['figure.subplot.top'] = .9

def _load_image(self, imfile): self.im = mplimage.imread(imfile)

@staticmethod
def _get_chno(ch):
chmap = {'R': 0, 'G': 1, 'B': 2}
return chmap.get(ch, -1)
def show_channel(self, ch):
bins = 256
ec = 'none'
chno = self._get_chno(ch)
loc = (chno, 1)
ax = plt.subplot2grid(self.shape, loc)
ax.hist(self.im[:, :, chno].flatten(), bins, color=ch, ec=ec,\ label=ch, alpha=.7)
ax.set_xlim(0, 255)
plt.setp(ax.get_xticklabels(), visible=True)
plt.setp(ax.get_yticklabels(), visible=False)
plt.setp(ax.get_xticklines(), visible=True)
plt.setp(ax.get_yticklines(), visible=False)
plt.legend()
plt.grid(True, axis='y')
return ax

def show(self):
loc = (0, 0)
axim = plt.subplot2grid(self.shape, loc, rowspan=3) axim.imshow(self.im)
```

```
plt.setp(axim.get_xticklabels(), visible=False) plt.setp(axim.get_yticklabels(),
visible=False) plt.setp(axim.get_xticklines(), visible=False)
plt.setp(axim.get_yticklines(), visible=False) axr = self.show_channel('R')
axg = self.show_channel('G')
axb = self.show_channel('B')
plt.show()

if __name__ == '__main__':
im = 'images/yellow_flowers.jpg' try:

iv = ImageViewer(im)
iv.show()
except Exception as ex:
print ex
```

How it works...

Reading from the end of the code, we see hard-coded filenames. These can be swapped by loading the argument from command line and parsing the given argument into the im variable using the sys.argv sequence.
We instantiate the ImageViewer class with the provided path to an image file. During object instantiation, we try to load an image file into an array, configure the figure via the rcParams dictionary, set the figure size and title, and define object fields (self.shape) to be used inside object's methods.

The main method here is show(), which creates a layout for the figure and loads the image arrays into the main (left column) subplot. We hide any ticks and tick labels as this is the actual image, where we don't have to use the ticks.

We then call the private method show_channel() for each of the red, green, and blue channels. This method also creates new subplot axes, this time in the right-hand side column, with each one in separate row. We plot the histogram for each channel in a separate subplot.

We also set up a little plot to remove unnecessary x ticks and add a legend in case we want to print this figure in a non-color environment. Therefore, we could discern channel representation even in those environments.

After we run this code we will get the following screenshot:

There's more...

The use of histogram plot type is just a choice for this image viewer example. We could have used any of the matplotlib supported plot types. Another real-world example would be to plot EEG or similar medical records where we would want to display slice as an image, the time series of EEG recorded as a line plot, and also additional meta information about the data shown, that would probably go into matplotlib.text.Text artists.

Having the ability to interact with the user GUI event, matplotlib's figure allows us also to implement interaction where we would want to zoom into all plots if we manually zoom on one plot only. That would be another usage where we want to display an image and zoom into it while also zoom into other displayed plots in the currently active figure. An idea would be to use motion_notify_event to call a function that will update x and y limits for all axes (subplots) in the current figure.

Plotting data on a map using Basemap

Probably the best geospatial visualizations are done by overlaying the data over the map. Whether the whole globe, a continent, a state, or even the sky, it is one of the easiest ways for a viewer to comprehend the relation between the data and geography it has displayed.

In this recipe we will be learning how to project data on a map using matplotlib's Basemap toolkit.

Getting ready

As we are already familiar with matplotlib as our plotting engine, we can extend that to matplotlib's capabilities to use other toolkits, one such example being the Basemap mapping toolkit.

Basemap itself doesn't do any plotting. It just transforms given geospatial coordinates to map projection and gives that data to matplotlib for plotting.

First, we need to install the Basemap toolkit. If you are using EPD, Basemap is already installed. If you are on Linux, it is best to use native package managers to install the package containing Basemap. On Ubuntu, for example, the package is called python-mpltoolkits. basemap and can be installed using standard package manager:

$ sudo apt-get install python-mpltoolkits.basemap

On Mac OS X it is recommended to use EPD, although installation using popular package managers such as Homebrew, Fink, and pip is also possible.

How to do it...

Here is a example on how to use the Basemap toolkit to plot simple Mercator projection within a specific region, specified by long, lat coordinate pairs:

1. We instantiate Basemap defining the projection to be used (merc for Mercator). 2. We define (in the same Basemap constructor) longitude and latitude for the lower-left and upper-right corners of a map.

3. We set up the Basemap instance map, to draw coastlines and countries.
4. We set up the Basemap instance map to fill continents and draw the map boundary.
5. We instruct the Basemap instance map to draw meridians and parallels.

The following code shows how to use Basemap toolkit to plot a simple Mercator projection:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np

map = Basemap(projection='merc',
resolution = 'h',
area_thresh = 0.1,

llcrnrlon=-126.619875, llcrnrlat=31.354158, urcrnrlon=-59.647219,
urcrnrlat=47.517613)

map.drawcoastlines()
map.drawcountries()
map.fillcontinents(color='coral', lake_color='aqua')
map.drawmapboundary(fill_color='aqua')

map.drawmeridians(np.arange(0, 360, 30)) map.drawparallels(np.arange(-90, 90,
30))
plt.show()
```

This will give a recognizable portion of our globe:

Now that we know how to plot a map, we need to know how to plot data on top of this map. If we recall that Basemap is a big transcoder of longitude and latitude pairs into current map projections, we recognize that all we need is a dataset that contains long/lat that we pass to Basemap for projecting, before plotting over with matplotlib. We use thecities.shp and cities.shx files to load the

coordinates of US cities and project them on to the map. The file is provided in the ch06 folder of the code repository. Here's the example on how to achieve this:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np

map = Basemap(projection='merc',
resolution = 'h',
area_thresh = 100,

llcrnrlon=-126.619875, llcrnrlat=25,
urcrnrlon=-59.647219, urcrnrlat=55)
shapeinfo = map.readshapefile('cities','cities')
x, y = zip(*map.cities)
# build a list of US cities
city_names = []
for each in map.cities_info:

if each['COUNTRY'] != 'US': city_names.append("")
else:
city_names.append(each['NAME'])

map.drawcoastlines()
map.drawcountries()
map.fillcontinents(color='coral', lake_color='aqua')
map.drawmapboundary(fill_color='aqua')
map.drawmeridians(np.arange(0, 360, 30))
map.drawparallels(np.arange(-90, 90, 30))

# draw city markers
map.scatter(x,y,25, marker='o',zorder=10)
# plot labels at City coords.
for city_label, city_x, city_y in zip(city_names, x, y): plt.text(city_x, city_y,
city_label)
plt.title('Cities in USA')
plt.show()
```

How it works...

The basics of Basemap usage consists of importing the main module and instantiating a Basemap class with desired properties. What we must specify during instantiations are the projections to be used and the portion of a globe that we want to work with.

Additional configuration can be applied before drawing the map and displaying the figure window with matplotlib.pyplot.show().

More than a dozen (or 32, to be precise) of different projections are supported in Basemap. Most of them are very narrow-usage oriented, but some are more general and applied for most common map visualizations.
We can easily see what projections are available by asking the Basemap module itself:

```
In [5]: import mpl_toolkits.basemap

In [6]: print mpl_toolkits.basemap.supported_ projections
mbtfpq McBryde-Thomas Flat-Polar Quartic
aeqd Azimuthal Equidistant
sinu Sinusoidal
poly Polyconic
omerc Oblique Mercator
gnom Gnomonic
moll Mollweide
lcc Lambert Conformal
tmerc Transverse Mercator
nplaea North-Polar Lambert Azimuthal
gall Gall Stereographic Cylindrical
npaeqd North-Polar Azimuthal Equidistant
mill
merc
stere
eqdc
cyl
npstere
spstere
hammer
```

geos
nsper
eck4
aea
kav7
spaeqd South-Polar Azimuthal Equidistant
ortho Orthographic
cass Cassini-Soldner
vandg van der Grinten
laea Lambert Azimuthal Equal Area
splaea South-Polar Lambert Azimuthal
robin Robinson
Miller Cylindrical Mercator Stereographic Equidistant Conic
Cylindrical Equidistant North-Polar Stereographic
South-Polar Stereographic
Hammer Geostationary Near-Sided Perspective Eckert IV Albers Equal Area
Kavrayskiy VII

Usually, we will plot the whole projection, if nothing is specified, some reasonable defaults are used.

To zoom in on a specific region of the map, we will specify the latitude and longitude of the lower-left and upper-right corners of the region you want to show. For this example, we will use the Mercator projection.

Here we see how the arguments names are shortened descriptions: f llcrnrlon: This is lower-left corner longitude f llcrnrlat: This is lower-left corner latitude f urcrnrlon: This is upper-right corner longitude f urcrnrlat: This is upper-right corner latitude

There's more...

We have just scratched the surface of the capabilities of Basemap toolkit, more examples can be found in the official documentation at http://matplotlib.org/basemap/users/ examples.html.

Most of the data used in the examples in the official Basemap documentation is located on remote servers and in a specific format. To efficiently fetch this data, NetCDF data format is used. NetCDF is a common data format designed with

network efficiency in mind. It allows a program to fetch as much data as is needed, even when the whole dataset is very large, which makes using this format very practical. We don't have to download and store large datasets locally every time we want to use them and every time they change.

Plotting data on a map using Google Map API

In this recipe, we will diverge from the desktop environment and show how we can output for the Web. Although, the main language for the web frontend is not Python but HTML, CSS, and JavaScript, we can still use Python for heavy lifting: fetch data, process it, perform intensive computations, and render data in a format(s) suitable for web output, that is, create HTML pages with the required JavaScript version to render our visualization(s).

Getting ready

We will use Google Data Visualization Library for Python to help us prepare data for the frontend interface, where we will use another Google Visualization API to render data in the desired visualization, that is, a map and a table.

Before we start, we need to install the google-visualization-python module. Download the latest stable version from https://code.google.com/p/google-visualizationpython/downloads/detail?name=gviz_api_py-1.8.2.tar.gz&can=2&q=, unpack the archive and install the module. The following actions demonstrate how to do this:

```
$ tar xfv gviz_api_py-1.8.2.tar.gz
$ cd gviz_api_py
$ sudo python ./setup.py install
```

On Windows and Mac OS X use the appropriate software to unpack the tar.gz archive, while the other steps should remain the same. Note that we have to become a super user (that is, gain administrator privileges) to install this module on our system.

A better option, if you don't want to pollute your OS packages, is to create a virtualenv environment to install the packages just for this recipe. We explained how to deal with virutalenv environments in *Chapter 1*, *Preparing Your Working Environment*.

For the frontend library we don't have to install anything, as that library will be loaded from the web page directly from the Google servers.

We need active access to Internet for this recipe because the output of it will be a web page that will, when opened in a web browser, pull the JavaScript libraries directly from remote servers.

In this recipe, we will be learning how to use Google Data Visualization Library for Python and JavaScript to combine them for creating web visualization.

How to do it...

The following example shows how to visualize Disposable Median Monthly Salary per Country on the world map projection using Google Geochart and Table Visualization, loading the data from a CSV file using Python and the gdata_viz module. We will:

1. Implement a function to act as a template generator.
2. Use the csv module to load the data from the local CSV file.

3. Use DataTable to describe the data and LoadData to load the data from the Python dictionary.
4. Render the output to a web page.

This can be achieved with the following code:

```
import csv
import gviz_api

def get_page_template():
page_template = """
<html>

<script src="https://www.google.com/jsapi" type="text/ javascript"></script>
<script>
google.load('visualization', '1', {packages:['geochart', 'table']});

google.setOnLoadCallback(drawMap); function drawMap() {
var json_data = new google.visualization.DataTable(%s, 0.6);
var options = {colorAxis: {colors: ['#eee', 'green']}}; var mymap = new
google.visualization.GeoChart( document.getElementById('map_div'));
mymap.draw(json_data, options);
```

```
var mytable = new google.visualization.Table(
document.getElementById('table_div'));
mytable.draw(json_data, {showRowNumber: true}) }
</script>
<body>
<H1>Median Monthly Disposable Salary World Countries</H1>

<div id="map_div"></div> <hr />
<div id="table_div"></div>

<div id="source">
<hr />
<small>
Source:
<a href="http://www.numbeo.com/cost-of-living/prices_by_

country.jsp?displayCurrency=EUR&itemId=105">
http://www.numbeo.com/cost-of-living/prices_by_country.jsp?dis
playCurrency=EUR&itemId=105
</a>
</small>
</div>
</body>
</html>
"""
return page_template

def main():
# Load data from CVS file
afile = "median-dpi-countries.csv" datarows = []
with open(afile, 'r') as f:

reader = csv.reader(f)
reader.next() # skip header for row in reader:

datarows.append(row)
# Describe data
description = {"country": ("string", "Country"), "dpi": ("number", "EUR"), }

# Build list of dictionaries from loaded data data = []
```

```
for each in datarows:

data.append({"country": each[0],
"dpi": (float(each[1]), each[1])})
# Instantiate DataTable with structure defined in 'description' data_table =
gviz_api.DataTable(description)
# Load it into gviz_api.DataTable data_table.LoadData(data)
# Creating a JSon string
json = data_table.ToJSon(columns_order=("country", "dpi"),
order_by="country", )

# Put JSON string into the template
# and save to output.html
with open('output.html', 'w') as out:

out.write(get_page_template() % (json,))
if __name__ == '__main__': main()
```

This will produce the output.html file, which we can open in our favorite web browser. The page should look like the following screenshot:

How it works...

The main entry point here is our main() function. First we use the csv module to load our data. This data is obtained from the public website www.numbeo.com, and data is put in the CSV format. The final file is available in the repository for this chapter in the ch06 folder. To be able to use from Google Data Visualization Library, we need to describe the data to it. We describe data using the Python dictionaries, where we define the ID of the columns, their data type, and optional label. In the following example the data is defined in this constraint:

```
{"name": ("data_type", "Label")}:
description = {"country": ("string", "Country"),
"dpi": ("number", "EUR"), }
```

Then we need to fit our loaded CSV rows in this format. We will build a list of dictionaries in the data variable.

Now we have everything to instantiate our data_table with gviz_data.DataTable with the described structure. Then we load the data into it and output in the JSON format to our page_template.

The get_page_template()function contains the other part of this equation. It contains a client (frontend) code to produce an HTML web page and a JavaScript code to load Google Data Visualization Library from Google servers. The line that loads the Google's JavaScript API is:

```
<script src="https://www.google.com/jsapi"
type="text/javascript"></script>
```

After this, follows another pair of <script>...</script> tags that contains an additional setup. First we load Google Data Visualization Library and the required package—geochart and table:

```
google.load('visualization', '1', {packages:['geochart', 'table']});
```

Then we set up a function that will be called when the pages are loaded. This event in the web world is registered as onLoad, so callback is set up via setOnLoadCallback function: google.setOnLoadCallback(drawMap);

This defines that when a page is loaded, the google instance will call the custom function drawMap() that we defined. The drawMap function loads a JSON string into the JavaScript version of the DataTable instance:

var json_data = new google.visualization.DataTable(%s, 0.6); Following that, we create a geochart instance in an HTML element with the ID map_div: var mymap = new google.visualization.GeoChart(
document.getElementById('map_div'));

Draw the map using json_data and provided custom options:
mymap.draw(json_data, options);
Similarly, Google's JavaScript table is rendered below the map:

```
var mytable = new google.visualization.Table(
document.getElementById('table_div'));
mytable.draw(json_data, {showRowNumber: true})
```

We save this output as an HTML file that we can open in a browser. This is not so useful for the dynamic rendering of a web service. There is a better option for

this—to output the HTTP Response directly from Python, and thus build a background service responding to client web requests with JSON that client can load and render.

If you want to understand more on reading HTTP responses, please read more on HTTP Protocol and Response messages at http://en.wikipedia. org/wiki/Hypertext_Transfer_Protocol#Response_message.

We do this by replacing the ToJson() call with the ToJSonResponse() with the same signature. This call will respond with a proper HTTP response containing the payload—our JSON-ified data_table ready to be consumed by our JavaScript client.

There's more...

This, of course, is just one example of how we can combine Python as a backend language, sitting on our server, doing the data fetch and processing, while the frontend is left to the universal HTML/JavaScript/CSS set of languages. This enables us to provide interactive and dynamic interfaces with visualizations to a wide audience without requiring them to install anything (well, apart from a web browser, but that is usually installed on a computer or smartphone). Saying that, we must note that the quality of these outputs is not as high as that of matplotlib, whereas the strength of matplotlib lies in the high quality output.

To work more with the Web (and Python) you would have to learn more about the web technologies and languages used. This book does not cover such topics but does give an insight into how to achieve one possible solution using well known third-party libraries that produce pleasing web outputs, with as little web coding as possible.

More documentation is available on the Google Developer portal at https://developers. google.com/chart/interactive/docs/dev/gviz_api_lib.

Generating CAPTCHA images

Although this is not strictly data visualization in terms that we usually refer to, the ability to generate images using Python comes in handy in many cases, and this is one of them. In this recipe, we will be covering the generation of random images to tell humans and computers apart—CAPTCHA image.

Getting ready

CAPTCHA stands for Completely Automated Public Turing test to tell Computers and Humans Apart, and is trademarked by Carnegie Mellon University. This test is used to challenge computer programs (usually referred to as bots) that automatically fill various web forms that are primarily targeted to humans and that should not be automated. Usual examples are sign-up forms, login forms, surveys, and similar.

CAPTCHA itself can take various forms, but the most common form consists of a challenge where humans should read an image with distorted characters and numbers and type in the result in the related response field.

In this recipe, we will learn how to harness Python's Imaging Library to generate images, rendering lines and points, and also rendering text.

How to do it...

We will show what would be involved in creating a personal and simple CAPTCHA generator by performing the following steps:

1. Define size, text, font size, background color, and CAPTCHA length.
2. Pick random characters from the English alphabet.
3. Draw those on the image using defined font and colors.
4. Add some noise in the form of lines and arcs.
5. Return the image object to the caller together with the CAPTCHA challenge.
6. Show the generated image to the user.

The following code shows how to create a personal and simple CAPTCHA generator:

```
from PIL import Image, ImageDraw, ImageFont
import random
import string

class SimpleCaptchaException(Exception): pass
class SimpleCaptcha(object):
def __init__(self, length=5, size=(200, 100), fontsize=36, random_text=None,
random_bgcolor=None):

self.size = size
self.text = "CAPTCHA"
```

```
self.fontsize = fontsize
self.bgcolor = 255 self.length = length

self.image = None # current captcha image
if random_text:
self.text = self._random_text()
if not self.text:
raise SimpleCaptchaException("Field text must not be empty.")
if not self.size:
raise SimpleCaptchaException("Size must not be empty.")
if not self.fontsize:
raise SimpleCaptchaException("Font size must be defined.")
if random_bgcolor:
self.bgcolor = self._random_color()

def _center_coords(self, draw, font):
width, height = draw.textsize(self.text, font)
xy = (self.size[0] - width) / 2., (self.size[1] - height) / 2. return xy

def _add_noise_dots(self, draw):
size = self.image.size
for _ in range(int(size[0] * size[1] * 0.1)):

draw.point((random.randint(0, size[0]), random.randint(0, size[1])), fill="white")

return draw

def _add_noise_lines(self, draw):
size = self.image.size
for _ in range(8):

width = random.randint(1, 2)
start = (0, random.randint(0, size[1] - 1)) end = (size[0],
random.randint(0,size[1]-1)) draw.line([start, end], fill="white", width=width)

for _ in range(8):
start = (-50, -50)
end = (size[0] + 10, random.randint(0, size[1]+10)) draw.arc(start + end, 0, 360,
fill="white")
```

return draw
def get_captcha(self, size=None, text=None, bgcolor=None): if text is not None:
self.text = text

if size is not None:
self.size = size
if bgcolor is not None:
self.bgcolor = bgcolor

self.image = Image.new('RGB', self.size, self.bgcolor) # Note that the font file must be present
# or point to your OS's system font
# Ex. on Mac the path should be '/Library/Fonts/Tahoma.ttf' font = ImageFont.truetype('fonts/Vera.ttf', self.fontsize) draw = ImageDraw.Draw(self.image)
xy = self._center_coords(draw, font)
draw.text(xy=xy, text=self.text, font=font)

# Add some dot noise
draw = self._add_noise_dots(draw)
# Add some random lines
draw = self._add_noise_lines(draw)
self.image.show()
return self.image, self.text

def _random_text(self):
letters = string.ascii_lowercase + string.ascii_uppercase random_text = ""
for _ in range(self.length):

random_text += random.choice(letters)
return random_text

def _random_color(self):
r = random.randint(0, 255)
g = random.randint(0, 255)
b = random.randint(0, 255)
return (r, g, b)

if __name__ == "__main__":
sc = SimpleCaptcha(length=7, fontsize=36, random_text=True,

```
random_bgcolor=True)
sc.get_captcha()
```

This produces an image similar to the following:

How it works...

This example gives a process on how to use Python's imaging library to generate predefined images, to create a simple, yet effective CAPTCHA generator.

We wrapped the functionality into one class SimpleCaptcha, because it gives us a safe space for future development. We also created a custom SimpleCaptchaException to accommodate future exception hierarchies.

If you are writing anything more than trivial and quick and dirty scripts, it is always good to start writing and designing custom exception hierarchies for your domain, rather than using generic Python's standard exceptions. You will gain a lot in the readability and maintenance of the software.

Start reading from the main section, at the end of code listing, where we instantiate class giving settings of our future image as arguments to the constructor. Following that, we call the get_captcha method on the sc object. For this recipe's purposes, get_captcha shows the image object as a result, but we also return the image object to the potential caller of this method, so it could make use of the result. The usage can vary, the caller could either save the image on the file, or if this was a web application, return the image stream and written challenge to the client requesting this CAPTCHA.

The important thing to note is that, in order to finish the challenge-response process of the CAPTCHA test, we must return the CAPTCHA string generated on the image as text so that the caller can compare the user's response with the expected values.

The get_captcha method first verifies the input arguments, in order to override the class' defaults if the user provides custom values. After that, a new image object is instantiated by Image.new. This object is saved in self.image, where we use it to draw and write text. Having written the text to the image, we add the noise of randomly placed points and lines, as well as some arc segments.

These tasks are carried out by the _add_noise_points and_add_noise_lines methods. The first one loops a few times and adds a point to a random location on the image, not too close to the edges of the image, and the latter one draws lines from the left-hand side of the image to the right-hand side of the image.

There's more...

We constructed this class using some assumptions about its use. We assumed that user will just want to accept our default settings (that is, random seven characters on a random background color) and receive the result from it. That is the reasoning behind placing helper functions in the constructor to set random text and random background color. If the most frequent and effective usage is to always override configuration, then we want to remove these operations from the constructor and place them in separate calls.

For example, maybe a user wants to always use English words as the CAPTCHA challenge. If this is the case, we want to be able to just call a method to provide us with results like that. This method could be get_english_captcha and with the random logic of this constructor, we would then construct that method to pick random words from the provided English dictionary. On a Unix system, there is a common English dictionary inside /usr/share/ dict/words that we could use for this:

```
def get_english_captcha(self):
words = '/usr/share/dict/words'
with open(words, 'r') as wf:

words = wf.readlines()
aword = random.choice(words)
aword = aword.strip() # remove newline and spaces

return self.get_captcha(text=aword)
```

Overall, the example of CAPTCHA generation is not production quality and should not be used without adding more protection and randomness, such as letter rotation.

If you need to protect your web forms from bots, there are already third-party Python modules and libraries that you should reuse. There are even specialized modules built for the existing web frameworks.

There are event web services such as reCAPTCHA (http://www.google.com/recaptcha) with already proven Python module recaptcha-client (https://pypi.python.org/ pypi/recaptcha-client) that you can sign up and use. It does not require any imaging libraries because the image is pulled directly from the reCAPTCHA web service but it has other dependencies such as pycrypto. By using this web service and library, you are also helping books scanned using Optical Character Recognition (OCR) from the Google Books project or old editions of the New York Times. Read more on the reCAPTCHA website.

7

Using Right Plots to Understand Data

In this chapter we will cover the following recipes:
f Understanding logarithmic plots

f Understanding spectrograms
f Creating a stem plot
f Drawing streamlines of vector flow
f Using colormaps
f Using scatter plots and histograms
f Plotting the cross-correlation between two variables
f Importance of autocorrelation

Introduction

In this chapter we will focus more on understanding what we want to say with the data we are presenting, and how to say it effectively. We will present some new techniques and plots, but all will be underlined by an understanding of the information we want to convey to the user. Let's ask the question, "Why do we want to present information in this state?". This is the most important question that should be asked during the data exploration phase. If we miss the opportunity to understand the data and present it in a certain way, the viewer, then, is not going to understand the data correctly for sure.

Understanding logarithmic plots

More often than not, reading daily newspapers and similar articles, one can find charts that are used by media organizations to misrepresent the facts. One common example is using linear scales to create so called panic charts, where a constantly growing value is followed for a long period of time (years) and starting values are smaller than the latest one by several magnitudes. These values when visualized correctly would (and usually should) produce linear or almost linear charts, taking some panic out of the articles they illustrate.

Getting ready

With the logarithmic scale, the ratio of consecutive values is constant. This is important when we are trying to read log plots. With linear (arithmetic) scales, the constant is the distance between consecutive values. In other words, logarithmic plots have a constant distance in orders of magnitude. We will see this illustrated in the following plots. The code used to produce this figure is explained next.

As a general rule of thumb, logarithmic scales should be used for the following conditions: f When the data presented has values that span several orders of magnitude f When the data presented has skewness towards large values (some points are much larger than the rest of the data)
f When you want to show the rate of change (growth rate), and not the value of change

Don't go blindly following these rules; they are more like hints, than rules. Always use your own judgment about the data in hand and requirements presented to you by the project or customer.

Depending on the data range, different log bases should be used. The standard base for logs is 10, but if the range of the data is smaller, a base of 2 can prove to be more useful as it will show more resolution within the smaller range.

If we have the range of data suitable for display on logarithmic scales, we will notice that the values previously being too close to judge any difference are now well apart. This allows us to read the chart much easier than if we would present the data in linear scale.

The growth rate charts, where long range time-series data is collected, are where we want to see not the absolute value measured at time point, but the growth in time. We will still get the absolute value information, but that information is of lower priority.

Also, if the data distribution has a positive skew, for example, salaries, taking the logarithm of the value (salary) will help us fit the data into the model, as the logarithm transformation will give us a more normal data distribution.

How to do it...

We will exemplify this with a sample code that shows the same two datasets (one linear, and one logarithmic in nature) on two different plots (in the same figure) using different scales (linear and logarithmic).

We will be performing the following steps with the help of the code mentioned after these steps:

1. Generate two simple datasets: y—exponential/logarithmic in nature and z—linear.
2. Create a figure containing a grid of four subplots.

3. Create two subplots containing the y dataset: one in logarithmic scale and one in linear scale.
4. Create two other subplots containing the z dataset, again one logarithmic and the other linear.
The following is the code:

```
from matplotlib import pyplot as plt
import numpy as np

x = np.linspace(1, 10) y = [10 ** el for el in x] z = [2 * el for el in x]

fig = plt.figure(figsize=(10, 8))

ax1 = fig.add_subplot(2, 2, 1)
ax1.plot(x, y, color='blue')
ax1.set_yscale('log')
ax1.set_title(r'Logarithmic plot of $ {10}^{x} $ ') ax1.set_ylabel(r'$ {y} =
{10}^{x} $')
plt.grid(b=True, which='both', axis='both')

ax2 = fig.add_subplot(2, 2, 2)
ax2.plot(x, y, color='red')
ax2.set_yscale('linear')
ax2.set_title(r'Linear plot of $ {10}^{x} $ ') ax2.set_ylabel(r'$ {y} = {10}^{x}
$') plt.grid(b=True, which='both', axis='both')

ax3 = fig.add_subplot(2, 2, 3) ax3.plot(x, z, color='green')
ax3.set_yscale('log')
ax3.set_title(r'Logarithmic plot of $ {2}*{x} $ ') ax3.set_ylabel(r'$ {y} = {2}*
{x} $')
```

```
plt.grid(b=True, which='both', axis='both')

ax4 = fig.add_subplot(2, 2, 4)
ax4.plot(x, z, color='magenta')
ax4.set_yscale('linear')
ax4.set_title(r'Linear plot of $ {2}*{x} $ ') ax4.set_ylabel(r'$ {y} = {2}*{x} $')
plt.grid(b=True, which='both', axis='both')

plt.show()
```

This code will produce the following output:

How it works...

We generate some sample data and two dependent variables: y and z. The variable y is expressed as an exponential function of x (data), and the variable z is a simple linear function of x. This helps us illustrate different looks of linear and exponential charts. We then create a grid of four subplots, where the top row subplots are of data (x, y) and bottom row are of data (x, z) pairs.

Looking from the left-hand side, the columns have logarithmic scales on the y axis, while from the right-hand side, the columns are in the linear scale. We set this using set_yscale('log') for every axis separately.

For every subplot we set the title and label, where the label also describes the

function plotted. With plt.grid(b=True, which='both', axis='both'), we turn the grid on for both the axes and both major and minor ticks.
We observe how linear functions are straight lines on linear plots, while logarithmic functions are straight lines on logarithmic plots.

Understanding spectrograms

A spectrogram is a time-varying spectral representation that shows how the spectral density of a signal varies with time.

A spectrogram represents a spectrum of frequencies of the sound or other signal in a visual manner. It is used in various science fields, from sound fingerprinting such as voice recognition, to radar engineering and seismology.

Usually, a spectrogram layout is as follows: the x axis represents time, the y axis represents frequency, and the third dimension is the amplitude of a frequency-time pair, which is color coded. This is three-dimensional data; therefore, we can also create 3D plots where the intensity is represented as the height on the z axis. The problem with 3D charts is that humans are bad at understanding and comparing them. Also, they tend to take more space than 2D charts.

Getting ready

For serious signal processing, we would go into low-level details to be able to detect patterns and autofingerprint certain specifics; but for this data visualization recipe we will leverage a couple of well-known Python libraries to read an audio file, sample it, and plot a spectrogram.

In order to read WAV files to visualize sound, we need to do some prep work. We need to install the libsndfile1 system library for reading/writing audio files. This is done via your favorite package manager. For Ubuntu, use:

$ sudo apt-get install libsndfile1-dev
It is important to install the dev package, which contains header files, so pip can build the scikits.audiolab module.

We also can install libasound and ALSA (Advanced Linux Sound Architecture) headers to avoid runtime warning. This is optional, as we are not going to use the features provided by the ALSA library. For Ubuntu Linux, issue the

following command:

$ sudo apt-get install libasound2-dev
To install scikits.audiolab, which we will use to read WAV files, we will use pip:
$ pip install scikits.audiolab

Always remember to enter the virtual environment for the current project, as you don't want to dirty the system libraries.

How to do it...

For this recipe, we will use the prerecorded sound file, test.wav, that can be found in the file repository of this book. But we could also generate a sample, which we will try later.

In this example, we perform the following steps in order:
1. Read the WAV file that contains a recorded sound sample.

2. Define the length of the window used for the Fourier Transform using NFFT.
3. Define the overlapping data points using noverlap while sampling.

```
import os
from math import floor, log

from scikits.audiolab import Sndfile
import numpy as np
from matplotlib import pyplot as plt

# Load the sound file in Sndfile instance
soundfile = Sndfile("test.wav")

# define start/stop seconds and compute start/stop frames start_sec = 0
stop_sec = 5
start_frame = start_sec * soundfile.samplerate stop_frame = stop_sec *
soundfile.samplerate

# go to the start frame of the sound object soundfile.seek(start_frame)

# read number of frames from start to stop delta_frames = stop_frame -
start_frame sample = soundfile.read_frames(delta_frames) map = 'CMRmap'

fig = plt.figure(figsize=(10, 6), )
```

```
ax = fig.add_subplot(111)
# define number of data points for FT
NFFT = 128
# define number of data points to overlap for each block noverlap = 65

pxx, freq, t, cax = ax.specgram(sample, Fs=soundfile.samplerate, NFFT=NFFT,
noverlap=noverlap, cmap=plt.get_cmap(map))

plt.colorbar(cax)
plt.xlabel("Times [sec]") plt.ylabel("Frequency [Hz]")

plt.show()
```

This generates the following spectrogram, with visible white-like traces for separate notes:

NFFT defines the number of data points used for computing the Discrete Fourier Transform in each block. The most efficient computation is when NFFT is a value raised to the power of 2. The windows can overlap and the number of data points that are overlapped (that is, repeated) is defined by the noverlap argument.

How it works...

We need to load a sound file first. To do that, we use the scikits.audiolab.SndFile method and provide it with a filename. This will instantiate a sound object, which we can then query for data and call the function on.

To read the data needed for the spectrogram, we need to read the desired frames of data from our sound object. This is done by read_frames(), which accepts the start and end frames. We calculate the frame number by multiplying the sample rate with the time points (start, end) we want to visualize.

There's more...

If you can't find audio (wave), you can easily generate one. Here's how to generate it:

```
import numpy
def _get_mask(t, t1, t2, lvl_pos, lvl_neg): if t1 >= t2:
raise ValueError("t1 must be less than t2") return
numpy.where(numpy.logical_and(t > t1, t < t2), lvl_pos, lvl_neg)

def generate_signal(t):
sin1 = numpy.sin(2 * numpy.pi * 100 * t) sin2 = 2 * numpy.sin(2 * numpy.pi *
200 * t)

# add interval of high pitched signal sin2 = sin2 * _get_mask(t, 2, 5, 1.0, 0.0)

noise = 0.02 * numpy.random.randn(len(t)) final_signal = sin1 + sin2 + noise
return final_signal
if __name__ == '__main__':

step = 0.001
sampling_freq=1000
t = numpy.arange(0.0, 20.0, step) y = generate_signal(t)

# we can visualize this now
# in time
ax1 = plt.subplot(211)
plt.plot(t, y)
# and in frequency
plt.subplot(212)
plt.specgram(y, NFFT=1024, noverlap=900,
```

```
Fs=sampling_freq, cmap=plt.cm.gist_heat)
plt.show()
```

This will give you the following signal where the top subplot represents the signal we generated. Here, the x axis represents time and the y axis represents the signal's amplitude. The bottom subplot represents the same signal in the frequency domain. Here, while the x axis represents time as in the top subplot (we matched the time by selecting the sampling rate), the y axis represents the frequency of the signal.

Creating a stem plot

A two-dimensional stem plot displays data as lines extending from a baseline along the x axis. A circle (the default) or other marker, whose y axis represents the data value, terminates each stem.

In this recipe we will be discussing how to create a stem plot.
Do not confuse stem with stem and leaf plots, which is a method of representing data by separating the last important digit of values as leaves and higher order values as stems.

```
steam | leaf
=========================
    0 | 6 7 8
    1 | 0 2 3 4 7 7 7 8 9
    2 | 1 3 4 4 5 7
    3 | 3 1 1 2 6 6 9
    4 | 1 5 5 6 9
    5 | 0
```

Getting ready

For this kind of plot we want to use a sequence of discrete data, where ordinary line plots will not make sense anyway.
Plot discrete sequences as stems, where data values are represented as markers at the end of each stem. Stems extend from the baseline (usually at y = 0) to the data point value.

How to do it...

We will use matplotlib to plot stem plots using the stem() function. This function can use just a series of y values when x values are generated as a simple sequence from 0 to len(y) - 1. If we provide the stem() function with both x and y sequences, they will be used for both axes.

What we want to configure with the stem plot is several formatters:

f linefmt: This is the line formatter for the stem line
f markerfmt: Stems at the end of the line are formatted using this argument f basefmt: This formats the look of the base line
f label: This defines the label for legend for the stem plot
f hold: This holds the current graphs on the current axes

f bottom: This sets up the location of baseline position on the y axis, and the default value is 0
The argument hold is used as a usual feature for plots. If it is on (True), all the following plots are added to the current axes. Otherwise, each plot will create new figures and axes.

To create a stem plot, perform the following steps:

1. Generate random noise data.

2. Configure stem options.
3. Plot the stem.

The following is the code to do it:

```
import matplotlib.pyplot as plt
import numpy as np
# time domain in which we sample x = np.linspace(0, 20, 50)
# random function to simulate sampled signal y = np.sin(x + 1) + np.cos(x ** 2)
# here we can setup baseline position bottom = -0.1

# True -- hold current axes for further plotting # False -- opposite. clear and use
new figure/plot hold = False

# set label for legend. label = "delta"
markerline, stemlines, baseline = plt.stem(x, y, bottom=bottom, label=label,
hold=hold)

# we use setp() here to setup
# multiple properties of lines generated by stem() plt.setp(markerline,
color='red', marker='o')
plt.setp(stemlines, color='blue', linestyle=':')
plt.setp(baseline, color='grey', linewidth=2, linestyle='-')

# draw a legend plt.legend()
plt.show()
```

The previous code produces the following plot:

How it works...

First, we need some data. For this recipe, the generated sampled pseudo-signal will suffice. In the real world, any discrete sequential data can be properly visualized using the stem plot. We generate this signal using Numpy's numpy.linspace, numpy.cos, and numpy.sin functions.

We then set up a label for the stem plot and the position of the baseline, which defaults to 0.0.
If we want to draw multiple stem plots, we will set hold toTrue and the resulting plot calls will be rendered over the same set of axes.

A call to matplotlib.stem returns three objects. First is markerline, an instance of Line2D. This holds the reference to a line representing stems themselves, rendering only markers and not the line connecting the markers. This line can be made visible by editing the property of that Line2D instance; the process for this will be explained soon. The final one is also a Line2D instance, baseline, holding

a reference to a horizontal line that represents the source of stemlines. The second object returned is stemlines, which is a collection (Python list at the moment) of Line2D instances representing stem lines, of course. We use these returned objects to manipulate the visual appeal of a stem plot using the setp function to apply properties to all the lines (the Line2D instances) in those objects, or a collections of objects.

Experiment with the desired settings until you understand how setp changes your plot's style.

Drawing streamlines of vector flow

Stream plots are used to visualize a flow in vector fields. Examples from Science and Nature include fields of magnetic and gravitational forces and movement of liquid materials.

A vector field can be visualized in such a way where we assign a line and one or more arrows to every point. The intensity can be represented by the line length, and the direction by an arrow pointing in a particular direction.

Usually, the intensity of the force is visualized with the length of a particular streamline, but the density can also be used for the same purpose.

Getting ready

To visualize vector fields, we will use matplotlib's matplotlib.pyplot.streamplot function. This function creates plots from streamlines of a flow, uniformly filling the domain. The velocities field is interpolated and streamlines are integrated. The original source for this function is to visualize wind patterns or liquid flow; hence, we don't need strict vector lines but a uniform representation of the vector field.

The most important arguments for this function are (X, Y), which are the evenly spaced grids of one-dimensional NumPy arrays, and (U, V), which match two-dimensional NumPy arrays of (X, Y) velocities. Matrices U and V must be of such dimensions that the number of rows must be of equal length as Y, and the number of columns must match the length of X.

The line width of the stream plot can be controlled per line, if the linewidth argument is given a two-dimensional array matching the shape of u and v

velocities, or it simply can be just one integer value that all the lines will accept.

Color, also, can be just one value for all streamlines and a matrix shaped like the linewidth argument.

Arrows (the class FancyArrowPatch) are used to indicate the vector direction and we can control them using two parameters: arrowsize to change the size of the arrow, and arrowstyle to change the format of the arrow (for example, "simple", "->"...).

How to do it...

We will start with a simple example, just to get a sense of what's going on here. Perform the following steps:

1. Create data vectors.
2. Print intermediate values.
3. Plot the stream plot.
4. Show the figure with streamlines visualizing our vectors.

The following is the code sample:

```
import matplotlib.pyplot as plt
import numpy as np
Y, X = np.mgrid[0:5:100j, 0:5:100j]
U = X
V = Y

from pprint import pprint
print "X"
pprint(X)

print "Y"
pprint(Y)
plt.streamplot(X, Y, U, V)
```

plt.show() The previous code will give the following textual output:

```
X
array([[ 0. , 0.05050505, 0.1010101 , ..., 4.8989899 ,

4.94949495, 5. ],
```

```
[ 0. , 0.05050505, 0.1010101 , ..., 4.8989899 ,
4.94949495, 5. ],
[ 0. , 0.05050505, 0.1010101 , ..., 4.8989899 ,
4.94949495, 5. ],
...,
[ 0. , 0.05050505, 0.1010101 , ..., 4.8989899 ,
4.94949495, 5. ],
[ 0. , 0.05050505, 0.1010101 , ..., 4.8989899 ,
4.94949495, 5. ],
[ 0. , 0.05050505, 0.1010101 , ..., 4.8989899 ,
4.94949495, 5. ]])
Y
array([[ 0. , 0. , 0. , ..., 0. ,
0. , 0. ],
[ 0.05050505, 0.05050505, 0.05050505, ..., 0.05050505,
0.05050505, 0.05050505],
[ 0.1010101 , 0.1010101 , 0.1010101 , ..., 0.1010101 ,
0.1010101 , 0.1010101 ],
...,
[ 4.8989899 , 4.8989899 , 4.8989899 , ..., 4.8989899 ,
4.8989899 , 4.8989899 ],
[ 4.94949495, 4.94949495, 4.94949495, ..., 4.94949495,
4.94949495, 4.94949495],
[ 5. , 5. , 5. , ..., 5. , 5. , 5. ]])
```

And it also generates the following streamline flow figure:

How it works...

We create a vector field of X and Y by indexing the two-dimensional mesh grid using NumPy's mgrid instance. We specify a range of the grid as start and stop (-2 and 2, respectively). The third index represents a step length. The step length represents the number of points to include between start and stop. If we want to include the stop value, we use a complex number for the step length, where the magnitude is used for a number of points required between start and stop, stop being inclusive.

The mesh grid, fleshed out like this, is then used to compute vector velocities. Here, for the sake of an example, we just use the same meshgrid property as vector velocities. This generates a plot that clearly shows plain linear dependency and the flow of the represented vector field.

Play with the values of U and V to get a sense of how the values of U and V influence the stream plot. For example, make U = np.sin(X) or V = np.sin(Y).

Following that, try to change the start and stop values. Check out the following plot for U = np.sin(X):

Bare in mind that the plot is a generated set of lines and arrow patches; hence, there is no way (currently, at least) to update the existing plot because lines and arrows know nothing about vectors and fields. Future implementations might include that, but at the moment this is a known limitation in the current version of matplotlib.

There's more...

Of course, this example gives just an opportunity to get to know and understand matplotlib's stream plot features and capabilities.

The real power comes when you have the real data at hand to play with. And after understanding this recipe, you will be able to recognize what are the tools you have, so that when you are given the data and you know its domain, you will

be able to pick the best tool for the job.

Using colormaps

Color coding the data can have a great impact on how your visualizations are perceived by the viewer, as they come with assumptions about color and what that color represents. Being explicit, if the color is used to add additional information to the data, it is always good. To know when and how to use color in your visualizations is even better.

Getting ready

If your data is not naturally color coded (such as earth/terrain altitudes or an object's temperature), it's better not to make any artificial mappings to natural coloring. We want to understand the data appropriately and so make a choice of color to help the reader decode data easily. We don't want readers constantly trying to suppress learned mapping of color for temperatures if we are representing financial data that has no connection with Kelvins or Celsius.

If possible, avoid usual red/green associations if there is no strong correlation in the data to associate them with those colors.

To help you pick the right color mapping, we will explain some colormaps available in the matplotlib package that can save a lot of time and help you, if you know what they are used for and how to find them.

Colormaps in general can be categorized as follows:

f Sequential: This represents monochromatic colormaps of two color tones from low to high saturation of the same color, for example, from white to bright blue. This is ideal for most cases, as they clearly show the change from low to high values.

f Diverging: This represents the central point and is the median value (some light color usually), but then ranges go to two different color tones in direction for high and low values. This can be ideal for data with a significant median value; for example, when the median is at 0, it clearly shows the difference between negative and positive values.

f Qualitative: For cases where data has no inherent ordering and all you want is

to make sure different categories are easily discernible from each other, this is the colormap to choose.

f Cyclic: This is handy to use where data can wrap around endpoint values, for example, representing time of the day, wind direction, or phase angle.

matplotlib comes with a lot of predefined maps, and we can divide them into several categories. We will suggest when to use some of those colormaps. The most common and base colormaps are autumn, bone, cool, copper, flag, gray, hot, hsv, jet, pink, prism, sprint, summer, winter, and spectral.

We have another set of colormaps coming from the Yorick scientific visualization package. This is an evolution from the GIST package, so all colormaps in this collection have gist_ as the prefix in their names.

The Yorick scientific visualization package is also an interpreted language written in C, not quite active lately. You can find more information on its official website at http://yorick.sourceforge.net/index.php.

These colormap sets contain the following maps: gist_earth, gist_heat, gist_ncar, gist_rainbow, and gist_stern.
Then we have the colormaps based on ColorBrewer (http://colorbrewer.org), where we can categorize them into the following:
f Diverging: This is where luminance is highest at the midpoint and decreases towards different endpoints
f Sequential: This is where luminance decreases monotonically
f Qualitative: This is where different sets of colors are used to differentiate data categories
Also, there are some miscellaneous colormaps that are available:

Colormap brg
bwr
coolwarm rainbow

seismic terrain

Description
This will represent a diverging blue-red-green colormap.
This will represent a diverging blue-white-red colormap.
This is useful for 3D shading, color blindness, and ordering of colors.

This represents a spectral purple-blue-green-yellow-orange-red colormap with diverging luminance.
This represents a diverging blue-white-red colormap.

This represents Mapmaker's colors—blue, green, yellow, brown, and white—originally from the IGOR Pro software.
Most of the maps presented here can be reversed by putting the _r postfix after the name of the colormap, for example, hot_r is an inversed cycle colormap of hot.

How to do it...

We can set colormaps on many items in matplotlib. For example, a colormap can be set on image, pcolor, and scatter. This is usually accomplished via an argument to a function call, cmap. This argument accepts an instance of colors.Colormap.

We can also use matplotlib.pyplot.set_cmap to set cmap for the latest object plotted on the axes.
You can get all the available colormaps easily with matplotlib.pyplot.colormaps.
Fire up IPython and type in the following:
In [1]: import matplotlib.pyplot as plt

In [2]: plt.colormaps() Out[2]:
['Accent',

'Accent_r',
'Blues',
'Blues_r',

...
'winter',
'winter_r']

Note that we have shortened the preceding list because it contains around 140 items and will span across several pages here.
This will import the pyplot function interface and allow us to call the colormaps function, which returns a list of all the registered colormaps.
Finally, we want to show you how to make a nice looking colormap. In the following example we need to:

1. Navigate to the ColorBrewer website to get divergent colormap color values in the Hex format.
2. Generate random samples of x and y, where y is the cumulative sum of values (simulate stock price variations).

3. Apply customization to the scatter plot function of matplotlib.
4. Tweak the scatter marker line color and width to make the plot more readable and pleasant for the viewer.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

# Red Yellow Green divergent colormap
red_yellow_green = ['#d73027', '#f46d43', '#fdae61', '#fee08b', '#ffffbf',
'#d9ef8b',
'#a6d96a', '#66bd63', '#1a9850']

sample_size = 1000
fig, ax = plt.subplots(1)

for i in range(9):
y = np.random.normal(size=sample_size).cumsum() x = np.arange(sample_size)
ax.scatter(x, y, label=str(i), linewidth=0.1,

edgecolors='grey',
facecolor=red_yellow_green[i])
ax.legend() plt.show()
```

The previous code will render a nice looking figure:

How it works...

We used the ColorBrewer website to find out colors in a red-yellow-green diverging colormap. Then, we listed those colors in our code and applied them to our scatter plot.

ColorBrewer is a web tool built by *Cynthia Brewer*, *Mark Harrower*, and *The Pennsylvania State University* to explore colormaps. It is a very handy tool to pick up colormaps of different ranges and see them applied on a map using slight variations, so that you immediately sense what they will look like on a chart. This particular map is at http://colorbrewer2.org/index.ph p?type=diverging&scheme=RdYlGn&n=9.

Sometimes, we will have to make our customization on matplotlib.rcParams, which is the first thing we want to do before we create a figure or any of the axes.
For example, matplotlib.rcParams['axes.cycle_color'] is the configuration setting we want to change in order to set up the default colormap for most of the matplotlib functions.

There's more...

Using matplotlib.pyplot.register_cmap, we can register a new colormap to matplotlib, so it can be found using the get_cmap function. We can use it in two different ways. Here are both the signatures:

```
f register_cmap(name='swirly', cmap=swirly_cmap)
f register_cmap(name='choppy', data=choppydata, lut=128)
```

The first signature allows us to specify a colormap as an instance of colors.Colormap and register it using the name argument. The argument name can be omitted in which case it will be inherited from the name attribute of the cmap instance provided.

For the latter one, we are passing three arguments to the linear-segmented colormap constructor and registering that colormap afterwards.
Using maplotlib.pyplot.get_cmap we can get the colors.Colormap instance using the name argument.
The following is how to make your own map using matplotlib.colors.LinearSegmentedColormap:

```
from pylab import *
cdict = {'red': ((0.0, 0.0, 0.0),
(0.5, 1.0, 0.7),
(1.0, 1.0, 1.0)),
'green': ((0.0, 0.0, 0.0),
(0.5, 1.0, 0.0),
(1.0, 1.0, 1.0)),
'blue': ((0.0, 0.0, 0.0),
(0.5, 1.0, 0.0),
(1.0, 0.5, 1.0))}
my_cmap = matplotlib.colors.LinearSegmentedColormap('my_
colormap',cdict,256)
pcolor(rand(10,10),cmap=my_cmap)
colorbar()
```

Executing this method is the simplest part, while the hardest part is to actually come up with a combination of colors that are informative, that do not take any information away from the data we want to visualize, and that are also pleasant to the eyes of the viewer.

For the base map list (the colormaps listed in the table earlier), we can use the pylab shortcut to set the colormap. For example:

```
imshow(X)
hot()
```

This would set the colormap of the image X to cmap='hot'.

Using scatter plots and histograms

Scatter plots are very often encountered, as they are the most common plot to visualize the relation between two variables. If we want to have a quick look at the data of these two variables and see if there is any relation between them (that is, correlation), we will draw a quick scatter plot. For a scatter plot to exist, we must have one variable that can be systematically changed by, for example, experimenter, so we can inspect the possibilities of influencing another variable.

That's why, in this recipe, we will learn how to understand the scatter plots.

Getting ready

We want to see, for example, how two events are affected by each other or if they are affected at all. This visualization is especially useful on large sets of data, where we cannot make any conclusions by looking at the data in the native form, when it is just numbers.

Correlation between values, if there is any, can be positive and negative. Positive correlation is for increasing X values, we have Y values increasing too. In negative correlation for increasing X values, Y values are decreasing. In an ideal case, positive correlation is a line starting from the bottom-left corner of the axes to the top-right corner. An ideal negative correlation is a line starting from the top-left corner to the bottom-right corner of the axes.
An ideal positive correlation between two data points is given the value of 1, and an ideal negative is given the value of -1. Everything inside this interval represents a weaker correlation between the two values. Usually, everything inside -0.5 to 0.5 is not considered valuable from the perspective of two variables being in real connection.

An example of positive correlation would be the amount of money put in a charity jar being directly positively correlated to the number of people seeing the jar. Negative correlation is between the time required to reach place B from place A, depending on the distance between the locations A and B. The bigger the distance, more will be the time we need to complete the travel.

The example we have presented here is of a positive correlation but this is not perfect, as different people might put different amounts of money per visit. But

in general, we can assume that the more the number of people who see the jar, more will be the money left inside.

Keep in mind, though, that even if the scatter plot displays a correlation between two variables, that correlation might not be a direct one. There might be a third variable that influences both the plotted variables, so the correlation is just a case that plotted values are correlated with that third variable. In the end, the correlation might be just apparent and no real relation exists behind it.

How to do it...

With the following code sample, we will demonstrate how the scatter plot can explain the relation between variables.
The data we use is obtained using the Google Trends web portal, where one can download the CSV file containing normalized values of relative search volumes for the given parameters. We will store our data in the ch07_search_data.py Python module, so we can import it in subsequent code recipes. The following is the content of it:

```
# ch07_search_data
# daily search trend for keyword 'flowers' for a year

DATA = [
1.04, 1.04, 1.16, 1.22, 1.46, 2.34, 1.16, 1.12, 1.24, 1.30, 1.44,
1.22, 1.26,
1.34, 1.26, 1.40, 1.52, 2.56, 1.36, 1.30, 1.20, 1.12, 1.12, 1.12,
1.06, 1.06,
1.00, 1.02, 1.04, 1.02, 1.06, 1.02, 1.04, 0.98, 0.98, 0.98, 1.00,
1.02, 1.02,
1.00, 1.02, 0.96, 0.94, 0.94, 0.94, 0.96, 0.86, 0.92, 0.98, 1.08,
1.04, 0.74,
0.98, 1.02, 1.02, 1.12, 1.34, 2.02, 1.68, 1.12, 1.38, 1.14, 1.16,
1.22, 1.10,
1.14, 1.16, 1.28, 1.44, 2.58, 1.30, 1.20, 1.16, 1.06, 1.06, 1.08,
1.00, 1.00,
0.92, 1.00, 1.02, 1.00, 1.06, 1.10, 1.14, 1.08, 1.00, 1.04, 1.10,
1.06, 1.06,
1.06, 1.02, 1.04, 0.96, 0.96, 0.96, 0.92, 0.84, 0.88, 0.90, 1.00,
1.08, 0.80,
0.90, 0.98, 1.00, 1.10, 1.24, 1.66, 1.94, 1.02, 1.06, 1.08, 1.10,
1.30, 1.10,
```

1.12, 1.20, 1.16, 1.26, 1.42, 2.18, 1.26, 1.06, 1.00, 1.04, 1.00,
0.98, 0.94,
0.88, 0.98, 0.96, 0.92, 0.94, 0.96, 0.96, 0.94, 0.90, 0.92, 0.96,
0.96, 0.96,
0.98, 0.90, 0.90, 0.88, 0.88, 0.88, 0.90, 0.78, 0.84, 0.86, 0.92,
1.00, 0.68,
0.82, 0.90, 0.88, 0.98, 1.08, 1.36, 2.04, 0.98, 0.96, 1.02, 1.20,
0.98, 1.00,
1.08, 0.98, 1.02, 1.14, 1.28, 2.04, 1.16, 1.04, 0.96, 0.98, 0.92,
0.86, 0.88,
0.82, 0.92, 0.90, 0.86, 0.84, 0.86, 0.90, 0.84, 0.82, 0.82, 0.86,
0.86, 0.84,
0.84, 0.82, 0.80, 0.78, 0.78, 0.76, 0.74, 0.68, 0.74, 0.80, 0.80,
0.90, 0.60,
0.72, 0.80, 0.82, 0.86, 0.94, 1.24, 1.92, 0.92, 1.12, 0.90, 0.90,
0.94, 0.90,
0.90, 0.94, 0.98, 1.08, 1.24, 2.04, 1.04, 0.94, 0.86, 0.86, 0.86,
0.82, 0.84,
0.76, 0.80, 0.80, 0.80, 0.78, 0.80, 0.82, 0.76, 0.76, 0.76, 0.76,
0.78, 0.78,
0.76, 0.76, 0.72, 0.74, 0.70, 0.68, 0.72, 0.70, 0.64, 0.70, 0.72,
0.74, 0.64,
0.62, 0.74, 0.80, 0.82, 0.88, 1.02, 1.66, 0.94, 0.94, 0.96, 1.00,
1.16, 1.02,
1.04, 1.06, 1.02, 1.10, 1.22, 1.94, 1.18, 1.12, 1.06, 1.06, 1.04,
1.02, 0.94,
0.94, 0.98, 0.96, 0.96, 0.98, 1.00, 0.96, 0.92, 0.90, 0.86, 0.82,
0.90, 0.84,
0.84, 0.82, 0.80, 0.80, 0.76, 0.80, 0.82, 0.80, 0.72, 0.72, 0.76,
0.80, 0.76,
0.70, 0.74, 0.82, 0.84, 0.88, 0.98, 1.44, 0.96, 0.88, 0.92, 1.08,
0.90, 0.92,
0.96, 0.94, 1.04, 1.08, 1.14, 1.66, 1.08, 0.96, 0.90, 0.86, 0.84,
0.86, 0.82,
0.84, 0.82, 0.84, 0.84, 0.84, 0.84, 0.82, 0.86, 0.82, 0.82, 0.86,
0.90, 0.84,
0.82, 0.78, 0.80, 0.78, 0.74, 0.78, 0.76, 0.76, 0.70, 0.72, 0.76,
0.72, 0.70,
0.64]

We need to perform the following steps: 1. Use a clean dataset of the Google Trend search volume for one year for the keyword flowers; we will import this dataset into the variable d.
2. Use a random normal distribution of the same length (365 data points) as our Google Trend dataset; this will be the dataset d1.

3. Create a plot containing four subplots.
4. In the first subplot, plot a scatter plot of d and d1.
5. In the second subplot, plot a scatter plot of d1 with d1.
6. In the third subplot, render a scatter plot of d1 with inverted d1.

7. And in the fourth subplot, render a scatter plot of d1 with a similar dataset constructed with a combination of d1 and d.
The following code will illustrate the relation as explained earlier in this recipe:

```
import matplotlib.pyplot as plt
import numpy as np
# import the data
from ch07_search_data import DATA
d = DATA

# Now let's generate random data for the same period d1 =
np.random.random(365)
assert len(d) == len(d1)

fig = plt.figure()

ax1 = fig.add_subplot(221) ax1.scatter(d, d1, alpha=0.5) ax1.set_title('No
correlation') ax1.grid(True)

ax2 = fig.add_subplot(222)
ax2.scatter(d1, d1, alpha=0.5)
ax2.set_title('Ideal positive correlation')

ax2.grid(True)

ax3 = fig.add_subplot(223)
ax3.scatter(d1, d1*-1, alpha=0.5) ax3.set_title('Ideal negative correlation')
ax3.grid(True)

ax4 = fig.add_subplot(224)
```

```
ax4.scatter(d1, d1+d, alpha=0.5)
ax4.set_title('Non ideal positive correlation') ax4.grid(True)

plt.tight_layout()
plt.show()
```

The following is the output we should get when the preceding code is executed:

How it works...

The sample we see in the preceding output clearly displays if there is any correlation between different datasets. While the second (top-right) subplot shows an ideal, or perfect, positive correlation of the dataset d1 with d1 itself (obviously), we can see that the fourth subplot (bottom right) hints that there is a positive correlation, although not ideal. We constructed this dataset from d1 and d (random) to simulate two similar signals (events), while the second subplot plotted using d and d1 has certain randomness (or noise) in it, but still can be compared with the original (d) signal.

There's more...

We can also add histograms to scatter plots in such a way that they can tell us more about the data plotted. We can add horizontal and vertical histograms to show frequencies of data points on the x and y axes. Using this, we can at the same time see the summary of the whole dataset (histogram) and individual data

points (the scatter plot).

The following is an example of the code to generate the scatter-histogram combination using the same two datasets we introduced in this recipe. The meat of the code is the function scatterhist() that is given here for reuse with different datasets, trying to set some of the variables based on the dataset provided (the number of bins in histogram, limits for axes, and so on).

We start with the usual imports:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
```

The following code is the definition of our function to generate scatter histograms given an (x, y) dataset and optionally, a figsize parameter:

```
def scatterhist(x, y, figsize=(8,8)):
"""
Create simple scatter & histograms of data x, y inside given plot

@param figsize: Figure size to create figure
@type figsize: Tuple of two floats representing size in inches
@param x: X axis data set
@type x: np.array

@param y: Y axis data set
@type y: np.array
"""

_, scatter_axes = plt.subplots(figsize=figsize)

# the scatter plot:
scatter_axes.scatter(x, y, alpha=0.5) scatter_axes.set_aspect(1.)

divider = make_axes_locatable(scatter_axes)
axes_hist_x = divider.append_axes(position="top", sharex=scatter_ axes, size=1,
pad=0.1)
axes_hist_y = divider.append_axes(position="right", harey=scatter_axes, size=1,
pad=0.1) # compute bins accordingly binwidth = 0.25
```

```
# global max value in both data sets
xymax = np.max([np.max(np.fabs(x)), np.max(np.fabs(y))]) # number of bins
bincap = int(xymax / binwidth) * binwidth

bins = np.arange(-bincap, bincap, binwidth) nx, binsx, _ = axes_hist_x.hist(x,
bins=bins, histtype='stepfilled',
orientation='vertical') ny, binsy, _ = axes_hist_y.hist(y, bins=bins,
histtype='stepfilled',
orientation='horizontal')

tickstep = 50
ticksmax = np.max([np.max(nx), np.max(ny)]) xyticks = np.arange(0, ticksmax
+ tickstep, tickstep)

# hide x and y ticklabels on histograms for tl in axes_hist_x.get_xticklabels():
tl.set_visible(False)
axes_hist_x.set_yticks(xyticks)

for tl in axes_hist_y.get_yticklabels(): tl.set_visible(False)
axes_hist_y.set_xticks(xyticks)

plt.show()
```

Now we proceed with loading of the data and function call to generate and render the desired chart.

```
if __name__ == '__main__': # import the data
from ch07_search_data import DATA as d

# Now let's generate random data for the same period d1 =
np.random.random(365)
assert len(d) == len(d1)

# try with the random data # d = np.random.randn(1000) # d1 =
np.random.randn(1000)

scatterhist(d, d1)
```

The previous code should generate the following output:

Plotting the cross-correlation between two variables

If we have two different datasets from two different observations, we want to know if those two event sets are correlated. We want to cross correlate them and see if they match in any way. We are looking for a pattern of a smaller data sample in a larger data sample. The pattern does not have to be an obvious or trivial pattern.

Getting ready

We can use matplotlib's matplotlib.pyplot.xcorr function from the pyplot lab. This function can plot the correlation between two datasets in such a way that we can see if there is any significant pattern between the plotted values. It is assumed that x and y are of the same length.

If we pass the argument normed as True, we can normalize by cross-correlation at 0th lag (that is, when there is no time delay or time lag).
Behind the scenes, correlation is done using NumPy's numpy.correlate function.

Using the argument usevlines (setting it to True), we can instruct matplotlib to use vlines() instead of plot() to draw the lines of the correlation plot. The main difference is if we are using plot(), we can style the lines using the standard Line2D properties passed in the **kwargs argument to the matplotlib.pyplot.xcorr function.

How to do it...

In this following example we need to perform the following steps:

1. Import the matplotlib.pyplot module.
2. Import the numpy package.
3. Use a clean dataset of the Google search volume trend for a year for the keyword flowers.
4. Plot the datasets (the real and artificial ones) and the cross-correlation diagram.
5. Tighten the layout in order to have a better overview of labels and ticks.
6. Add appropriate labels and grids for easier understanding of the plot.

The following is the code that will perform the previously mentioned steps:

```
import matplotlib.pyplot as plt
import numpy as np
# import the data
from ch07_search_data import DATA as d

total = sum(d)
av = total / len(d)
z = [i - av for i in d]

# Now let's generate random data for the same period d1 =
np.random.random(365)
assert len(d) == len(d1)
```

```
total1 = sum(d1)
av1 = total1 / len(d1) z1 = [i - av1 for i in d1]

fig = plt.figure()

# Search trend volume
ax1 = fig.add_subplot(311)
ax1.plot(d)
ax1.set_xlabel('Google Trends data for "flowers"')

# Random: "search trend volume" ax2 = fig.add_subplot(312) ax2.plot(d1)
ax2.set_xlabel('Random data')

# Is there a pattern in search trend for this keyword? ax3 = fig.add_subplot(313)
ax3.set_xlabel('Cross correlation of random data')
ax3.xcorr(z, z1, usevlines=True, maxlags=None, normed=True, lw=2)
ax3.grid(True)
plt.ylim(-1, 1)

plt.tight_layout()
plt.show()
```

The previous code will render the following output:

How it works...

We used a real dataset with a recognizable pattern in it (two peaks repeating in a similar manner across the dataset; refer to the preceding plot). The other dataset is just some random normal-distributed data of the same length as the real accrued data from the public service, Google Trends.

We plotted both datasets over the top half of the output to visualize the data. Using matplotlib's xcorr, which in turn uses NumPy's correlate() function, we computed cross-correlation and plotted it on the bottom half of the screen.

Cross-correlation computation in NumPy returns a correlation coefficient's array that represents a degree of similarity of two datasets (or signals, as usually referred to if used in the signal processing field).

The cross-correlation diagram, correlogram, tells us that these two signals are not correlated, which is represented by the height of the correlation values (vertical lines that appear at certain time lags). We can see that there is more than one vertical line (the correlation coefficient at time lag *n*) that is above 0.5.

If, for example, two datasets would have the correlation at a time lag of 100 (that is, a 100-second shift between the same object observed by two different sensors), we will see a vertical line (representing the correlation coefficient) at x = 100 in the preceding output.

Importance of autocorrelation

Autocorrelation represents the degree of similarity between a given time series and a lagged (that is, delayed in time) version of itself over successive time intervals. It occurs in time series studies when the errors associated with a given time period carry over into future time periods. For example, if we are predicting the growth of stock dividends, an overestimate in one year is likely to lead to overestimates in the succeeding years.

The time series analysis data arises in lots of different scientific applications and financial processes. Some of the examples include: generated reports of financial performance, prices over time, computing volatility, and others.

If we are analyzing unknown data, autocorrelation can help us detect whether the data is random or not. For that we can use correlogram. It can help provide answers to questions such as: Is the data random? Is this time series data a white

noise signal? Is it sinusoidal? Is it autoregressive? What is the model of this time series data?

Getting ready

We will use matplotlib to compare two sets of data. One is the Google day trend of search volume for a certain keyword for one year (365 days). The other set is 365 random measurements (the generated random data) with normal distribution.

We will autocorrelate both datasets and compare how the correlograms visualize patterns in data.

How to do it...

In this section we will perform the following steps:

1. Import the matplotlib.pyplot module.
2. Import the numpy package.
3. Use a clean dataset of Google search volume for a year.
4. Plot the dataset and plot its autocorrelation diagram.
5. Generate a same-length random dataset using NumPy.
6. Plot the random dataset on the same figure, and plot its autocorrelation diagram.
7. Add appropriate labels and grids for easier understanding of the plot.

The following is the code:

```
import matplotlib.pyplot as plt
import numpy as np
# import the data
from ch07_search_data import DATA as d

total = sum(d)
av = total / len(d)
z = [i - av for i in d]

fig = plt.figure()
# plt.title('Comparing autocorrelations')

# Search trend volume
ax1 = fig.add_subplot(221)
ax1.plot(d)
ax1.set_xlabel('Google Trends data for "flowers"')
```

```
# Is there a pattern in search trend for this keyword? ax2 = fig.add_subplot(222)
ax2.acorr(z, usevlines=True, maxlags=None, normed=True, lw=2)
ax2.grid(True)
ax2.set_xlabel('Autocorrelation')

# Now let's generate random data for the same period d1 =
np.random.random(365)
assert len(d) == len(d1)

total = sum(d1)
av = total / len(d1) z = [i - av for i in d1]

# Random: "search trend volume" ax3 = fig.add_subplot(223) ax3.plot(d1)
ax3.set_xlabel('Random data')

# Is there a pattern in search trend for this keyword? ax4 = fig.add_subplot(224)
ax4.set_xlabel('Autocorrelation of random data')
ax4.acorr( z, usevlines=True, maxlags=None, normed=True, lw=2)
ax4.grid(True)

plt.show()
```

The previous code will render the following output:

How it works...

Looking at the left-hand side plots, it is easy to spot patterns in the search volume data, where the bottom-left plot has normally distributed random data with patterns that are not obvious, but still might exist.

Computing and plotting autocorrelation over random data, we can see that there is a high correlation at 0, which is expected, and data is correlated with itself without any time lag. But going before or after no time lag, the signal is almost 0. So we can safely conclude that there is no correlation between the signal in original time and any time lags examined.

Looking at the real data, Google search volume trend, we can see the same behavior at zero time lag, still something we can expect for any autocorrelated signal. But we have strong signals at around 30, 60, and 110 days after zero time lag. This indicates that there is a pattern with this particular search term and the way people search for it on the Google search engine.

We will leave the exercise of explaining why is this a very different story to the reader. Remember that correlation and causation are two very different things.

There's more...

Autocorrelation is used very often when we want to identify a model for unknown data. When we try to fit data into a model, how data correlates to itself is sometimes the first step to identifying the appropriate model for a dataset we are presented with. This requires more than Python; it requires knowledge of mathematical modeling and various statistical tests (LjungBox test, Box-Pierce test, and so on) that will help us answer any questions we may have.

8

More on matplotlib Gems

In this chapter we will cover:
f Drawing barbs

f Making a box and whisker plot
f Making Gantt charts
f Making errorbars
f Making use of text and font properties
f Rendering text with LaTeX
f Understanding the difference between pyplot and OO API

Introduction

In this chapter we will explore some less frequently used features of the matplotlib package. Some of these examples stretch the matplotlib original target, but they show what can be done with a little creativity, and prove that matplolib is full featured and generically oriented.

Drawing barbs

A barb is a representation of the speed and direction of wind, and is mainly deployed by meteorology scientists. In theory they can be used to visualize any type of two-dimensional vector quantities. They are similar to arrows (quivers), but the difference is that arrows represent vector magnitude by the length of the arrow, while barbs give more information about the vector's magnitude by employing lines or triangles as increments of magnitude. We will explain what barbs are, how to read them, and how to visualize them using Python and matplotlib. Here's a typical set of barbs:

0 5 10 15 30 40 50 60 100
In the preceding diagram, the triangle, also known as flag, represents the largest increment. A full line or barb, represents a smaller increment; a half line is the smallest increment.

The increments are in the order of 5, 10, and 65 for a half-line, line, and triangle respectively. The values here represent, for meteorologists at least, wind speed in nautical miles per hour (knots).

We ordered the barbs from left to right to represent the following magnitudes: 0, 5, 10, 15, 30, 40, 50, 60, and 100 knots. The direction here is the same for each barb and is from north to south, because the east-west speed component is 0 for each barb.

Getting ready

A barb can be created using a matplotlib function from matplotlib.pyplot.barbs.

The barbs function accepts various arguments, but the main use case is that we specify X and Y coordinates, representing locations of observed data points. The second pair of arguments—U, V—represents the magnitude of the vector in north-south and east-west directions in knots.

Other arguments that can be useful are pivots, sizes, and various coloring arguments.

A pivot argument (pivot) represents the part of the arrow represented on the grid point. We get a pivot argument when the arrow rotates around this point. The arrow can rotate around the tip or middle, which are valid values for the pivot argument.

Because barbs consist of several parts, we can set up the coloring of any of those parts. So we have a few color-related arguments that we can set up:

f barbcolor: This defines the color of all the parts for a barb, except for flags f flagcolor This defines the color of any flag on the barb

f facecolor: This argument is used if none of the preceding color arguments are specified (or the default value is read from rcParams)

If any of the preceding color-related arguments are specified, the argument facecolor is overridden. The argument facecolor is the one used in coloring polygons. The size argument (sizes) specifies the ration of a feature to the length of the barb. This is a collection of coefficients that can be specified by using any or all of the following keys:

f spacing: This defines the space among features of the flag/barb
f height: This defines the distance from the shaft to the top of a flag or barb f width: This defines the width of a flag
f emptybarb: This defines the circle radius used for low magnitudes

How to do it...

Let's demonstrate how to use a barb function by performing the following steps:

1. Generate a grid of coordinates to simulate observations.
2. Simulate observational values for wind speed.
3. Plot barb diagrams.
4. Plot quivers to demonstrate different appearances.

The following code will generate the figure:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-20, 20, 8)
y = np.linspace( 0, 20, 8)
# make 2D coordinates
X, Y = np.meshgrid(x, y)
U, V = X+25, Y-35

# plot the barbs
plt.subplot(1,2,1)
plt.barbs(X, Y, U, V, flagcolor='green', alpha=0.75) plt.grid(True, color='gray')

# compare that with quiver / arrows
plt.subplot(1,2,2)
plt.quiver(X, Y, U, V, facecolor='red', alpha=0.75)

# misc settings
plt.grid(True, color='grey') plt.show()
```

The preceding code renders two subplots as shown in the following figure:

How it works...

To illustrate how the same data can bring different information to light, we used barbs and quiver plots from matplotlib to visualize simulated observed wind data.

First, we used NumPy to generate samples of variations for x and y arrays. Then we used NumPy's meshgrid() function to create a 2D grid of coordinates where our observed data is sampled at certain coordinates. Finally, U and V are wind speed values in NS (north-south) and EW (east-west) directions, in knots (nautical miles per hours). For the purpose of the recipe, we adjusted some values from the already available X and Y matrices.

We then divided the figure into two subplots, plotting barbs in the leftmost plot and arrow-patches in the rightmost plot. We adjusted the color and transparency of both the subplots slightly, as well as turned the grid on both the subplots.

There's more...

This is all fine on the northern hemisphere where the wind rotates in a counter-clockwise direction and the feathers (triangles, full lines and half lines of the barb) point in the direction of lower pressure. On the southern hemisphere, this is inverted so our wind barb graph would not represent the data we are visualizing correctly.
We have to invert this direction of feathers. Luckily, the barbs function has the

argument flip_barb. This argument can be of one single Boolean value (True or False) or a sequence of Boolean values such as the shape of other data arrays, when each item in the sequence specifies a flip decision for each barb.

Making a box and a whisker plot

Do you want to visualize a series of data measurement (or observations) to show several properties of the data series (such as the median value, the spread of the data, and the distribution of the data) in one plot? And would you want to do that in a way where you can visually compare several similar data series? How would you visualize them? Welcome to the box-and-whisker plot! Probably the best plot type for comparing distributions, if you are talking to people used to information density.

The box-and-whisker plot usage examples range from comparing test scores between schools to comparing process parameters before and after changes (optimization).

Getting ready

What are the elements of box and whisker plots? As we see in the following diagram, we have several important elements that carry information in the box-and-whisker plot. The first component is the box that carries information about the interquartile range going from lower to upper quartile values. The median value of the data is represented by a line across the box.

The whiskers extend from the box on both sides going from the first quartile (25 percentile) to the last quartile (75 percentile) of the data. In other words, the whiskers extend 1.5 times from the base of the inter-quartile range. In the case of a normal distribution, whiskers will cover 99.3 percent of the total data range.

If there are values outside the whiskers range, they will be displayed as fliers. Otherwise, the whiskers will cover the total range of the data.
Optionally, the box can also carry information about confidence intervals around the median. This is represented by a notch in the box. This information can be used to indicate whether the data in the two series is of the similar distribution. However, this is not rigorous and is just an indication that can be visually inspected.

How to do it...

In the following recipe we will learn how to create a box-and-whisker plot using matplotlib. We will perform the following steps:
1. Sample some comparative process data, where a single integer number represents the occurrence of an error during the observed period of the running process.

2. Read data from the PROCESSES dictionary into DATA.
3. Read labels from the PROCESSES dictionary into LABELS.
4. Render the box-and-whisker plot using matplotlib.pyplot.boxplot.
5. Remove some chart junk from the figure.
6. Add axes labels.
7. Show the figure.

The following code implements these steps:

```
import matplotlib.pyplot as plt
# define data
PROCESSES = {

"A": [12, 15, 23, 24, 30, 31, 33, 36, 50, 73],
"B": [6, 22, 26, 33, 35, 47, 54, 55, 62, 63],
"C": [2, 3, 6, 8, 13, 14, 19, 23, 60, 69],
"D": [1, 22, 36, 37, 45, 47, 48, 51, 52, 69],
}

DATA = PROCESSES.values() LABELS = PROCESSES.keys()
plt.boxplot(DATA, notch=False, widths=0.3)
# set ticklabel to process name plt.gca().xaxis.set_ticklabels(LABELS)
# some clean up(removing chartjunk) # turn the spine off
for spine in plt.gca().spines.values(): spine.set_visible(False)

# turn all ticks for x-axis off
plt.gca().xaxis.set_ticks_position('none') # leave left ticks for y-axis on
plt.gca().yaxis.set_ticks_position('left')

# set axes labels
plt.ylabel("Errors observed over defined period.") plt.xlabel("Process observed
over defined period.")

plt.show()
```

The preceding code generates the following figure:

How it works...

The box and whisker plot is rendered by first computing quartiles for the given data in DATA. These quartile values are used to compute lines to draw boxes and whiskers.

We adjusted the plot to be more visually pleasing and not contain any unnecessary lines (referring to superfluous lines such as "chart junk", as mentioned in the famous book *The Visual Display of Quantitative Information* by *Edward R. Tufte*). Those lines do not carry information and just put more pressure on the mental models in a viewer's brain to decode all the lines before discovering real valuable information.

Making Gantt charts

One form of very widely-used visualization of time-based data is a Gantt chart. Named after the mechanical engineer *Henry Gantt* who invented it in 1910s, it is

almost exclusively used to visualize work breakdown structures in project management. This chart is loved by managers for its descriptive value and not so loved by employees, especially when the project deadline is near.

Because it is very common, almost every one can understand and read it, even if it is overloaded with additional (related and unrelated) information.

A basic Gantt chart has a time series on the x axis, and a set of labels that represent tasks or subtasks on the y axis. Task duration is usually visualized either as a line or as a bar chart, extending from the start to end time of a given task.

If subtasks are present, one or many subtasks have a parent task, in which the case total time of a task is aggregated from subtasks in such a way that overlapping and gap time is accounted for. This is useful to perform critical path analysis.

Critical path analysis is a mathematical analysis that computes a path containing all the tasks required, taking into account tasks interdependencies so that the total start to completion time of a project can be calculated. This is a very important tool in the project management field and can be universally applied to any type of project for scheduling and resource planning.

So, in this recipe we will be covering the creation of the Gantt chart using Python.

Getting ready

There are many full-fledged software applications and services that allow you to make very flexible and complicated Gantt charts. We will try to demonstrate how you could do it in pure Python, not relying on external applications, yet achieving neat looking and informative Gantt charts.
The Gantt chart shown in the example does not support nested tasks, but it is sufficient for simple work breakdown structures.

How to do it...

The following code example will allow us to demonstrate how Python can be used together with matplotlib to render the Gantt chart. We will perform the following steps: 1. Load TEST_DATA that contains a set of tasks and instantiate the Gantt class with TEST_DATA.

2. Each task contains a label and the start and end time.
3. Process all tasks by plotting horizontal bars on the axes.
4. Format x and y axes for the data we are rendering.
5. Tighten the layout.
6. Show the Gantt chart.
The following is a sample code:

```
from datetime import datetime
import sys

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.font_manager as font_manager
import matplotlib.dates as mdates

import logging

class Gantt(object):
'''
Simple Gantt renderer.
Uses *matplotlib* rendering capabilities. '''

# Red Yellow Green diverging colormap # from http://colorbrewer2.org/
RdYlGr = ['#d73027', '#f46d43', '#fdae61',

'#fee08b', '#ffffbf', '#d9ef8b', '#a6d96a', '#66bd63', '#1a9850']
POS_START = 1.0

POS_STEP = 0.5 def __init__(self, tasks):
self._fig = plt.figure()
self._ax = self._fig.add_axes([0.1, 0.1, .75, .5])

self.tasks = tasks[::-1]

def _format_date(self, date_string):
'''
Formats string representation of *date_string* into

*matplotlib.dates*
instance.
```

'''
try:

date = datetime.strptime(date_string, '%Y-%m-%d %H:%M:%S') except
ValueError as err:
logging.error("String '{0}' can not be converted to datetime object: {1}"
.format(date_string, err))
sys.exit(-1)
mpl_date = mdates.date2num(date)
return mpl_date

def _plot_bars(self):
'''
Processes each task and adds *barh* to the current *self._ax*

(*axes*).
'''
i = 0
for task in self.tasks:

start = self._format_date(task['start'])
end = self._format_date(task['end'])
bottom = (i * Gantt.POS_STEP) + Gantt.POS_START width = end - start
self._ax.barh(bottom, width, left=start, height=0.3,

align='center', label=task['label'], color = Gantt.RdYlGr[i])
i += 1

def _configure_yaxis(self):
'''y axis'''
task_labels = [t['label'] for t in self.tasks] pos = self._positions(len(task_labels))
ylocs = self._ax.set_yticks(pos)
ylabels = self._ax.set_yticklabels(task_labels) plt.setp(ylabels, size='medium')

def _configure_xaxis(self): '''x axis'''
make x axis date axis self._ax.xaxis_date()

# format date to ticks on every 7 days
rule = mdates.rrulewrapper(mdates.DAILY, interval=7) loc =
mdates.RRuleLocator(rule)

```
formatter = mdates.DateFormatter("%d %b")

self._ax.xaxis.set_major_locator(loc)
self._ax.xaxis.set_major_formatter(formatter) xlabels =
self._ax.get_xticklabels()
plt.setp(xlabels, rotation=30, fontsize=9)

def _configure_figure(self): self._configure_xaxis() self._configure_yaxis()

self._ax.grid(True, color='gray') self._set_legend()
self._fig.autofmt_xdate()

def _set_legend(self):
'''
Tweak font to be small and place *legend* in the upper right corner of the figure
'''
font = font_manager.FontProperties(size='small') self._ax.legend(loc='upper
right', prop=font)

def _positions(self, count):
'''
For given *count* number of positions, get array for the

positions.
'''
end = count * Gantt.POS_STEP + Gantt.POS_START pos =
np.arange(Gantt.POS_START, end, Gantt.POS_STEP) return pos
```

The main function that drives the Gantt chart generation is defined in the following code. In this function, we load the data into an instance, plot bars accordingly, set up the date formatter for the time axis (x axis), and set values for the y axis (the project's tasks).

```
def show(self):
self._plot_bars()
self._configure_figure()
plt.show()

if __name__ == '__main__': TEST_DATA = (
```

{ 'label': 'Research', 'start':'2013-10-01
12:00:00', 'end': '2013-10-02 18:00:00'}, # @IgnorePep8 { 'label': 'Compilation', 'start':'2013-10-02
09:00:00', 'end': '2013-10-02 12:00:00'}, # @IgnorePep8 { 'label': 'Meeting #1', 'start':'2013-10-03
12:00:00', 'end': '2013-10-03 18:00:00'}, # @IgnorePep8 { 'label': 'Design', 'start':'2013-10-04
09:00:00', 'end': '2013-10-10 13:00:00'}, # @IgnorePep8 { 'label': 'Meeting #2', 'start':'2013-10-11
09:00:00', 'end': '2013-10-11 13:00:00'}, # @IgnorePep8 { 'label': 'Implementation', 'start':'2013-10-12
09:00:00', 'end': '2013-10-22 13:00:00'}, # @IgnorePep8 { 'label': 'Demo', 'start':'2013-10-23
09:00:00', 'end': '2013-10-23 13:00:00'}, # @IgnorePep8)

gantt = Gantt(TEST_DATA) gantt.show()

This code will render a simple, neat looking Gantt chart like the following one:

How it works...

We can start reading the preceding code from the bottom after the condition that checks if we are in "__main__".

After we instantiate the Gantt class giving it TEST_DATA, we set up the necessary fields of our instance. We save TASK_DATA in the self.tasks field, and we create our figure and axes to hold the charts we create in future.

Then we call show() on the instance that walks us through the steps required to render the Gantt chart:

```
def show(self):
self._plot_bars()
self._configure_figure()
plt.show()
```

Plotting bars requires an iteration where we apply the data about the name and duration of each task to the matplotlib.pyplot.barh function, adding it to the axes at self._ax. We place each task in a separate channel by giving it a different (incremental) bottom argument value.

Also, to make it easy to map tasks to their names, we cycle over the divergent color maps that we generated using the colorbrewer2.org tool.

The next step is to configure the figure, which means that we set up the format date on the x axis and tickers' positions and labels on the y axis to match the tasks plotted by matplotlib.pyplot.barh.

We do a final tweaking of grid and legend.
At the end, we call plt.show() to show the figure.

Making errorbars

Error bars are useful to display the dispersion of data on a plot. They are relatively simple as a form of visualization; however, they are also a bit problematic because what is shown as an error varies across different sciences and publications. This does not lessen the usefulness of error bars, it just imposes the need to always be careful and explicitly state the nature of the error visualized as an error bar.

Getting ready

To be able to plot an error bar in the raw observed data, we need to compute the mean and the error we want to display.
The error we compute represents the 95 percent confidence interval that the mean we get from our observation is stable, which means our observations are good estimates of the whole population.

matplotlib supports these type of plots via matplotlib.pyplot.errorbar function. It offers several capabilities around error bars. They can be vertical (yerr) or

horizontal (xerr), and symmetrical or asymmetrical.

How to do it...

In the following code we will:

1. Use some sample data that consists of four sets of observations.

2. For each set of observations, compute the mean value.
3. For each set of observations, compute the 95 percent confidence interval.
4. Render bars with vertical symmetrical error bars.

Here is the code for this:

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as sc

TEST_DATA = np.array([[1,2,3,2,1,2,3,4,2,3,2,1,2,3,4,4,3,2,3,2,3,2,1],
[5,6,5,4,5,6,7,7,6,7,7,2,8,7,6,5,5,6,7,7,7,6,5],
[9,8,7,8,8,7,4,6,6,5,4,3,2,2,2,3,3,4,5,5,5,6,1],
[3,2,3,2,2,2,2,3,3,3,3,4,4,4,4,5,6,6,7,8,9,8,5], ])

# find mean for each of our observations
y = np.mean(TEST_DATA, axis=1, dtype=np.float64) # and the 95% confidence
interval
ci95 = np.abs(y - 1.96 * sc.sem(TEST_DATA, axis=1))

# each set is one try
tries = np.arange(0, len(y), 1.0)

# tweak grid and setup labels, limits
plt.grid(True, alpha=0.5)
plt.gca().set_xlabel('Observation #')
plt.gca().set_ylabel('Mean (+- 95% CI)')
plt.title("Observations with corresponding 95% CI as error bar.") plt.bar(tries, y,
align='center', alpha=0.2) plt.errorbar(tries, y, yerr=ci95, fmt=None)

plt.show()
```

The preceding code will render a plot with error bars that display 95 percent confidence intervals as whiskers extending along the y axis. Remember, the

wider the whiskers, the lesser is the probability that the observed mean is true. The following graph is the output for the preceding code:

How it works...

In order to avoid iterating over each set of observations, we use NumPy's vectorized methods to compute means and standard errors, which we use for plotting and computing error values. Using NumPy's vectorized implementations that are written in C language (and called from Python) allows us to speed up computations by several magnitudes.
This is not very important for a few data points but, for millions of data points, it can either make or break our efforts to create responsive applications.

Also, you may notice that we explicitly specified dtype=np.float 64 in the np.mean function call. According to the official NumPy documentation reference (http://docs. scipy.org/doc/numpy/reference/generated/numpy.mean.html), np.mean can be inaccurate if used in single precision; it's better to compute it with np.float32, or if performance is not an issue, use np.float 64.

There's more...

There is an ongoing issue with what to show on error bars. Some advise on using

SD, *2SD*, *SE*, or *95%CI*. We must understand what the difference between all these values and what they are used for, in order to be able to give reasoning on what to use and when.

Standard Deviation informs us about the distribution of individual data points around the mean value. If we assume normal distribution, then we know that *68.2% (~2/3)* of data values will fall between ±*SD*, and *95.4%* of values will be between ±*2*SD*.

Standard Error is calculated as *SD* divided by the square root of *N* (*SD/√N*), where *N* is the number of data points. Standard Error (SE) informs us about variability of mean values, if we are able to perform the same sampling more than once (like performing the same study hundreds of times).

The confidence interval is calculated from SE, similar to how the range of values is calculated from Standard Deviation. To calculate 95 percent confidence interval, we must add/subtract *1.96 * SE* to/from our mean value or use proper notation: *95% CI = M ± (1.96 * SE)*. The wider the confidence interval, the lesser we would be sure that we are right.

We see that in order to be sure that our estimation is correct and that we are giving its proof to our reader, we should display the confidence interval, which in turn carries the standard error; this, if small, proves that our means are stable.

Making use of text and font properties

We already learned how to annotate the plot by adding legends, but sometimes we want more with text. This recipe will explain and demonstrate more features of text manipulation in matplotlib, giving a powerful toolkit for even advanced typesetting needs.

We will not cover LaTeX support in this recipe, as there is a recipe named *Rendering text with LaTeX* in this chapter.

Getting ready

We start with listing of the most useful set of functions that matplotlib offers. Most of the functions are available via pyplot module's interface, but we map their origin function here to allow you to explore more if a particular text feature is not covered in this recipe. Basic text manipulations and their mapping in

matplotlib OO API is presented in the following table:

matplotlib.pyplot text
Matplotlib API
matplotlib.axes.Axes.text
xlabel matplotlib.axes.Axes.set_xlabel
ylabel matplotlib.axes.Axes.set_ylabel
title matplotlib.axes.Axes.set_title
suptitle matplotlib.figure.Figure.suptitle
figtext matplotlib.figure.Figure.text
Description

Adds text to the axes at the location specified by (x, y). Argument fontdict allows us to override generic font properties, or we can use kwargs to override specific property.

Sets the label for the x
axis. Specifies the spacing between the label and the x axis in accordance with labelpad.

Similar to xlabel, but
intended for the y axis.

Sets the title for the axes. Accepts all the usual
text properties such as
fontdict and kwargs.

Adds a centered title to the figure. Accepts all the usual text properties via kwargs. Uses figure coordinates.

Puts text anywhere on the figure. The location is defined using x,y, using figure's normalized coordinates. Override font properties using fontdict, but also support kwargs to override any text-related property.

The base class for text storing and drawing inside windows or data coordinates is the matplotlib.text.Text class. It supports the definition of the location of text objects as well as a range of properties that we can define, to tune how our strings are going to appear on a figure or a window.
The font properties supported by the matplotlib.text.Text instances are:

Property family
size or
fontsize
style or fontstyle
variant
weight or fontweight
Values

'serif',
'sans-serif', 'cursive',
'fantasy',
'monospace'
12, 10,... or 'xx-small', 'x-small',
'small',
'medium',
'large',
'x-large', 'xxlarge'
'normal',
'italic',
'oblique'
'normal',
'small-caps' 0-1000 or
'ultralight', 'light',
'normal',
'regular',
'book',
'medium',
'roman',
'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'

Description

Specifies the font name or font family. If this is a list, then it is ordered by priority, so the first matched name will be used.

Specifies the size in relative or absolute points, or specifies the relative size as a size string.
Specifies the font style as a string.
Specifies the font variant.

Specifies the font weight or using a specific weight string. Font weight is defined as the thickness of character outline relative to its height.
Property

stretch or fontstretch
Values

0-1000 or
'ultra
condensed', 'extra
condensed', 'condensed', 'semi
condensed', 'normal',
'semi
expanded', 'expanded', 'extra
expanded', 'ultra
expanded'

Description

Specifies the stretch of the font. Stretch is defined as horizontal condensation or expansion. This property is not currently implemented.

fontproperties Defaults to the matplotlib.font_manager.

FontProperties instance. This class stores and manages font properties as described in W3C CSS Level1 specification at http://www.w3.org/TR/1998/REC-CSS2-19980512/.

We can also specify the background box that will contain the text, and which can be further specified in color, borders, and transparency.
The basic text color is read from rcParams['text.color'], if not specified on the current instance, of course.
Specified text can also be aligned according to visual needs. There are the following alignment properties:

f horizontalalignment or ha: This allows alignment of text horizontally to center, left, and right.
f verticalalignment or va: Allowed values for this are center, top, bottom, and baseline.
f multialignment: This allows alignment of text strings that span multilines.

Allowed values are: left, right, and center.

How to do it...

So far all is good, but we have a hard time visualizing all these variations in the fonts we can create. So this is going to illustrate what we can do. In the next code we will be performing the following steps:

1. List all the possible properties we want to vary on the font.
2. Iterate over the first set of variations: font family and size.
3. Iterate over the second set of variations: weight and style.

4. Render text samples for both the iterations and print the variation combination as a text on the plot.
5. Remove axes from the figure, as they serve no purpose.

The following is the code:

```
import matplotlib.pyplot as plt
from matplotlib.font_manager import FontProperties

# properties:
families = ['serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'] sizes = ['xx-small',
'x-small', 'small', 'medium', 'large',

'x-large', 'xx-large']
styles = ['normal', 'italic', 'oblique']
weights = ['light', 'normal', 'medium', 'semibold', 'bold', 'heavy', 'black']
variants = ['normal', 'small-caps']

fig = plt.figure(figsize=(9,17))
ax = fig.add_subplot(111)
ax.set_xlim(0,9)
ax.set_ylim(0,17)

# VAR: FAMILY, SIZE
y = 0
size = sizes[0]
style = styles[0]
weight = weights[0]
variant = variants[0]
```

```
for family in families:
x = 0
y = y + .5
for size in sizes:

y = y + .4
sample = family + " " + size
ax.text(x, y, sample, family=family, size=size,

style=style, weight=weight, variant=variant) # VAR: STYLE, WEIGHT
y = 0
family = families[0]
size = sizes[4]
variant = variants[0]

for weight in weights: x = 5
y = y + .5
for style in styles:

y = y + .4
sample = weight + " " + style
ax.text(x, y, sample, family=family, size=size,

style=style, weight=weight, variant=variant)
ax.set_axis_off() plt.show()
```

The preceding code will produce the following screenshot:

monospace xx-large
monospace x-large
monospace large
monospace medium
monospace small
monospace x-small
monospace xx-small

fantasy xx-large
fantasy x-large
fantasy large
fantasy medium
fantasy small
fantasy x-small
fantasy xx-small

cursive xx-large
cursive x-large
cursive large
cursive medium
cursive small
cursive x-small
cursive xx-small

sans-serif xx-large
sans-serif x-large
sans-serif large
sans-serif medium
sans-serif small
sans-serif x-small
sans-serif xx-small

serif xx-large
serif x-large
serif large
serif medium
serif small
serif x-small
serif xx-small

black oblique
black italic
black normal

heavy oblique
heavy italic
heavy normal

bold oblique
bold italic
bold normal

semibold oblique
semibold italic
semibold normal

medium oblique
medium italic
medium normal

normal oblique
normal italic
normal normal

light oblique
light italic
light normal

How it works...

The code is really straightforward, as we just iterate twice over tuples of

properties printing their values.
The only trick employed here is the positioning of text on the figure canvas, as that allows us to have nice layout of text samples we can easily compare.

Keep in mind that the default font matplotlib will use is dependent on the operating system you are running, so the preceding screenshot might look slightly different. This screenshot was rendered using standard Ubuntu 13.04 installed fonts.

Rendering text with LaTeX

If we want to plot more scientific graphics and explain math as it should be using scientific notations and complex equations on the figures, we need support from the best. Although matplotlib has support for math text rendering, the best support comes from the LaTeX community, proven in the task being used for many decades.

LaTeX is a high-quality typesetting system for the production of scientific and technical documentation, being a *de facto* standard for scientific typesetting or publication. It is a free software, available on the majority of desktop platforms used today as prepackages binary installation; hence, it is easy to install.

The basic syntax of LaTeX is similar to markup languages; so to produce satisfactory content, one would write focusing more on the structure than on the look and style. For example:

```
\documentclass{article}
\title{This here is a title of my document}
\author{Peter J. S. Smith}
\date{September 2013}
\begin{document}

\maketitle
Hello world, from LaTeX!
\end{document}
```

We see how this is different from the usual word processor, where you have the WYSIWYG editor environment and the style is already applied to your text. Sometimes this is good but, for scientific publications, style is a secondary

concern; the primary focus is having the right, correct, and valid content. Here, by content we also mean mathematical notations (usually a lot of it), including graphs.

Apart from this, there are many more features such as automatic generation of bibliographies and indexes, which are important for medium to large publications. And those are the main focus points of the LaTeX system. Because this is not a book about LaTeX, we will stop with the quick introduction here. A lot more documentation is available on the project's website at http://latex-project.org/.

Getting ready

Before we start demonstrating matplotlib's support for rendering text using LaTeX, we need to have the following packages installed on our system:
f LaTeX system: The most common one is the TeX Live prepackaged distribution f DVI to PNG converter: This makes PNG graphics from DVI files as obtained from TeX, by producing anti-aliased screen-resolution images
f Ghost script: This is required, unless already installed by TeX Live distribution

There are different prepackaged systems of the LaTeX environment for different operating systems. For Linux-based systems, TeX Live is a complete TeX system. For Mac OS, the recommended environment is the MacTeX distribution; for the Windows environment, the proTeX system is going to install all the TeX supports, including LaTeX.

Whichever package you install, make sure it comes with font libraries and programs for typesetting, previewing, and printing of TeX documents in many different languages. We will install our package for Linux using the texlive and dvipng packages for Ubuntu. We can install this by using the following command:
$ sudo apt-get install texlive dvipng

The next step is to tell our matplotlib to use LaTeX by setting text.usetex toTrue. We can do that either in our custom .matplotlibrc inside our home directory (/home/<user>/. matplotlibrc on Unix-based systems, or C:\Documents and Settings\<user>\. matplotlibrc) via rcParams['text'], or by using the following code:

```
matplotlib.pyplot.rc('text', usetex=True)
```

The start of the code will tell matplotlib to go back to LaTeX for all text rendering. It is important to do this before we add any figure and axis.
Not all backends support LaTeX rendering. Only the Agg, PS and PDF backends support text rendering via LaTeX.

How to do it...

What we want to do here is demonstrate the basic usage properties of LaTeX.
We will perform the following steps:

1. Generate some sample data.
2. Set up matplotlib to use LaTeX for this plotting session.
3. Set up the font and font properties to be used.
4. Write out the equation syntax.
5. Demonstrate the usage of Greek symbols' syntax.
6. Draw math notations of fractions and fractals.
7. Write some limits and exponential expressions.
8. Write possible range expressions.
9. Write expressions with text and formatted text in them.
10. Write some math expressions on x and y labels as figure titles.

The following code will perform these steps:

```
import numpy as np
import matplotlib.pyplot as plt

# Example data
t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(4 * np.pi * t) * np.sin(np.pi*t/4) + 2

plt.rc('text', usetex=True)
plt.rc('font',**{'family':'sans-serif','sans-serif':['Helvetica'], 'size':16})

plt.plot(t, s, alpha=0.25)

# first, the equation for 's'
# note the usage of Python's raw strings
plt.annotate(r'$\cos(4 \times \pi \times {t}) \times \sin(\pi \times \ frac {t} 4) +
2$', xy=(.9,2.2), xytext=(.5, 2.6), color='red', arrowpr ops={'arrowstyle':'->'})

# some math alphabet
plt.text(.01, 2.7, r'$\alpha, \beta, \gamma, \Gamma, \pi, \Pi, \phi, \ varphi, \Phi$')
# some equation
```

```
plt.text(.01, 2.5, r'some equations $\frac{n!}{k!(n-k)!} = {n \choose k}$')
# more equations
plt.text(.01, 2.3, r'EQ1 $\lim_{x \to \infty} \exp(-x) = 0$') # some ranges...
plt.text(.01, 2.1, r'Ranges: $( a ), [ b ], \{ c \}, | d |, \| e \|, \ langle f \rangle, \lfloor
g \rfloor, \lceil h \rceil$')
# you can multiply apples and oranges
plt.text(.01, 1.9, r'Text: $50 apples \times 100 oranges = lots of juice$')
plt.text(.01, 1.7, r'More text formatting: $50 \textrm{ apples} \times 100 \textbf{
apples} = \textit{lots of juice}$')
plt.text(.01, 1.5, r'Some indexing: $\beta = (\beta_1,\beta_2,\dotsc,\ beta_n)$')
# we can also write on labels
plt.xlabel(r'\textbf{time} (s)')
plt.ylabel(r'\textit{y values} (W)')
# and write titles using LaTeX
plt.title(r"\TeX\ is Number "

r"$\displaystyle\sum_{n=1}^\infty\frac{-e^{i\pi}}{2^n}$!", fontsize=16,
color='gray')
# Make room for the ridiculously large title.
plt.subplots_adjust(top=0.8)

plt.savefig('tex_demo') plt.show()
```

The preceding code will render the following text-saturated figure that demonstrates LaTeX rendering:

How it works...

After we set up the rendering engine and font properties, we basically used standard matplotlib calls for text rendering, such as matplotlib.pyplot.annotate, matplotlib. pyplot.text, matplotlib.pyplot.xlabel, matplotlib.pyplot.ylabel, and matplotlib.pyplot.title.

The difference here is that all the strings are so-called raw strings, meaning Python will not interpret them and no string substitution will occur; hence the LaTeX engine is going to receive exactly the same strings as commands to act upon.

More examples of the TeX syntax and how to use it in matplotlib can be found on official matplotlib documentation at http://matplotlib.org/users/mathtext. html#writing-mathematical-expressions.

Note that this URL is not on LaTeX but on matplotlib's own integrated TeX parser. This parser supports almost the same syntax, and it can even be sufficient for your needs.

There's more...

If you run into a problem while setting up this environment or have different problems with fonts that either look bad or are not able to produce the LaTeX rendering, make sure that you have installed all required packages, your $PATH environment variable (if on Windows) is set up to include all the required binaries, and matplotlib is set to use LaTeX for text rendering.

If all of the given instructions are followed and the results cannot be replicated, refer to the official matplotlib website at http://matplotlib.org/users/usetex.html#possible-hangups, and the LaTeX community on http://tex.stackexchange.com/ for further assistance.

It is known that this setup is not as streamlined as it should be, and some quirks may occur for various reasons.

Understanding the difference between pyplot and OO API

This recipe will try to explain some of the programming interfaces in matplotlib and make a comparison of pyplot and object-oriented API (Application Programming Interface). Depending on the task at hand, this will allow us to decide why and when to use either of those interfaces.

Getting ready

When the matplotlib library started, it was similar to many open source projects—there was no proper (free) solution to the problem a person had, so he wrote one. The problem encountered with MATLAB® was with performance for the task in hand (http://www.aosabook.org/ en/matplotlib.html), and the original author already had knowledge of both MATLAB® and Python, so he started writing matplotlib as a solution for his need for the current project.

This is the main reason matplotlib has a MATLAB®-like interface that allows one to quickly plot data without worrying about background details, which platform matplotlib is running on, which are the underlying rendering libraries (is it Linux with GTK, Qt, Tk, or wxWidgets either on Linux or Windows), or are we running on Mac OS with the help of Cocoa toolkits. This is all hidden inside matplotlib under a nice procedural interface in the matplotlib.pyplot module, a stateful interface handling logic for creating figures and axes to connect them with the configured backend. It also keeps data structures for the current figure and axes, which are called upon with plot commands.

This is the interface (matplotlib.pyplot) we were using throughout most of this book as it is simple, straightforward, and good enough for most of the tasks we were trying to accomplish. The matplotlib library was designed with this philosophy in mind. We must be able to draw plots with as few commands as possible, even just one command (for example, plt.plot([1,2,3,4,5]); plt.show() works!). For these tasks we don't want to be forced into thinking about objects, instances, methods, properties, rendering backends, figures, canvases, lines, and other graphical primitives.

If you are reading this book from the start, you probably noticed that some classes started appearing in various examples; such as, FontProperties or AxesGrid, where we needed more than is provided by the matplotlib.pyplot module.

This is the object-oriented programming interface that implements all the hidden hard stuff such as rendering graphical elements, rendering those to the platform's graphical toolkit, and handling user inputs (mouse and keystrokes). There is nothing to stop us from using OO API, and that is what we are going to do.

So if we take a look at matplotlib as a software, it consists of three parts:
f matplotlib.pylab interface: This is a set of functions for the user to create plots like in MATLAB®
f matplotlib API (also called matplotlib frontend): This is a set of classes for the creation and management of figures, text, lines, plots, and so on

f backends: These are drawing drivers; they transform front abstract representation into a file or a display device
This backend layer contains concrete implementations of abstract interface classes. There are classes such as FigureCanvas (a surface to draw onto paper), Renderer (a paintbrush that does the drawing on the canvas), and Event (a class that handles the user's keystrokes and mouse events).

The code is also separated. The base abstract classes are in matplotlib.backend_bases and every concrete implementation is in a separate module. For example, the GTK 3 backend is in matplotlib.backends.backend_gkt3agg.

In this stack there is an Artist classes' hierarchy where most of the hard stuff is done. Artist knows about Renderer and how to use it to draw images on

FigureCanvas. Most of the stuff we are interested in (text, lines, ticks, tick labels, images, and so on) are Artist or subclasses of the Artist class (located in the matplotlib.artist module).

The class, matplotlib.artist.Artist, contains all the shared properties of its children: coordinates transformation, clip box, label, user event handlers, and visibility.

In this figure Artist is the base for most of the other classes. There are two basic categories of classes that inherit from Artist. The first category is of primitive artists, which are visible objects such as Line2D, Rectangle, Circle, and Text. The second category is of composite artists, which are collections of other Artists such as Axis, Tick, Axes, and Figure. For example, Figure has the background of the primitive artist Rectangle, but also contains at least one composite artist, Axes.

Most of the plotting is happening on the Axes class (matplotlib.axes.Axes). Figure background elements such as ticks, axis lines, and the grid and color of the background patch is contained in Axes. Another important feature of Axes is that all the helper methods create other primitive artists and add them to the Axes instance; for example, plot, hist, and imshow.
Axes.hist, for example, creates many matplotlib.patch.Rectangle instances and stores them in the Axes.patches collection.

Axes.plot creates one or more matplotlib.lines.Line2D and stores them in the

Axes.lines collection.

How to do it...

As an illustration we will:

1. Instantiate the matplotlib Path object for custom drawing.
2. Construct the vertices of our object.
3. Construct the path's command codes to connect those vertices.
4. Create a patch.
5. Add it to the Axes instance of figure.

The following code implements our intentions:

```
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches

# add figure and axes
fig = plt.figure()
ax = fig.add_subplot(111)

coords = [
(1., 0.), # start position (0., 1.),
(0., 2.), # left side
(1., 3.),
(2., 3.),
(3., 2.), # top right corner (3., 1.), # right side (2., 0.),
(0., 0.), # ignored
]

line_cmds = [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO,
Path.LINETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.CLOSEPOLY,
]

# construct path
path = Path(coords, line_cmds)
# construct path patch
patch = patches.PathPatch(path, lw=1,

facecolor='#A1D99B', edgecolor='#31A354') # add it to *ax* axes
ax.add_patch(patch)
```

```
ax.text(1.1, 1.4, 'Python', fontsize=24) ax.set_xlim(-1, 4)
ax.set_ylim(-1, 4)
plt.show()
```

The preceding code will generate the following:

How it works...

For this octagon we used the base patch matplotlib.path.Path, which supports the basic set of primitives for drawing lines and curves (moveto and lineto). These can be used to draw simple and also more advanced polygons using Bezier curves.

First, we specified a set of coordinates in the data coordinates that we match with a set of path commands to act upon those coordinates (or vertices, if you like). With that we instantiate matplotlib.path.Path. We then construct the patch instance matplotlib. patched.PathPatch with that path, which is a general polycurve path patch.

This patch can now be added to the figure's axes (the fig.axes collection) and we can render the figure to show the polygon.

What we didn't want to do in this example is use matplotlib.figure.Figure directly in place of the matplotlib.pyplot.figure() call. The reason for this is that the pyplot. figure() call does a lot in the background, such as reading the rc parameters from the matplotlibrc file (to load default figsize, dpi and figure color settings), setting up the figure manager class (Gcf), and so on. We could do all that, but until we really know what we are doing, this is the recommended way to create the figure.

As a general rule of thumb, unless we cannot achieve something via the pyplot interface, we should not reach for direct classes such as Figure, Axes, and Axis, because there is a lot of state managing going on in the background; so, unless we are developing matplotlib, we should avoid bothering about that.

There's more...

If you want interactivity and exploration, it would be the best to use matplotlib via the Python interactive shell. For this purpose, probably the most well known is the IPython pylab mode. This gives you all the matplotlib features in a powerful and introspective shell with rich set features such as history, inline plotting, and the possibility to share your work if you use IPython Notebook.

IPython Notebook is a web-based interface to the IPython shell, where the work can be shared and converted into HTML or PDF. Matplotlib plots are embedded and inlined, so they can also be saved and shared.

Symbols

3D bars
creating 139-143
3D histograms
creating 143-146
3D visualization
about 139
3D bars, creating 139-143
3D histograms, creating 143-146
.rc file 14

A

Advanced Linux Sound Architecture (ALSA) 194
animate() function 149
animation
with OpenGL 150-154
animation, matplotlib 146-150
Animation (object) class 146
annotations
adding 92-94
array slicing 54
ArtistAnimation (TimedAnimation) class 147
autocorrelation
about 220
importance 220
plotting 221-223
Axes3D 140
AxesGrid 139
axis() function 82
axis label
size, setting 110-112
transparency, setting 110-112
axis lengths
defining 81-83

Index

axis limits
defining 81-83
B
backends 251
background color
defining 89

barb
about 225
drawing 225-229

barbs function 226

bar charts
creating 99-101

bard3d function 146
bar plot 72-78
Basemap

about 139
used, for plotting data on map 172-176

Basemap toolkit
URL, for documentation 177
box plot
about 77
making 229-232
brg colormap 206
bwr colormap 206

C

CAPTCHA image
about 183, 184
generating 183-188

cell() method 24
chart line
shadow, adding to 113-116
color
defining 88

ColorBrewer
URL 206
Colorbrewer2
URL 85
colored markers
used, for drawing scatter plots 105, 107
colormaps
using 205-210
colormaps, categories
cyclic 206

diverging 205
qualitative 206
sequential 205
colormaps, ColorBrewer
category 206
colormaps function 207
Comma Separated Values. *See* CSV
contour() function
about 125
call signature 126
contourf(..., V) call signature 126
contour plot
about 125
creating 125-128
contour(X,Y,Z) call signature 126
contour(X,Y,Z,N) call signature 126
contour(X,Y,Z,V) call signature 126
contour(Z) call signature 126
contour(Z, **kwargs) call signature 126
contour(Z,N) call signature 126
contour(Z,V) call signature 126
controlled random datasets
generating 56-64
convolve function 64
coolwarm colormap 206
Coordinate system
Axes 113
Data 113
Display 113
Figure 113
correlate() function 220
cosine plot
drawing 78-81
create_thumbnails() function 163
cross-correlation
plotting, between two variables 217-220 CSV 20
CSV file
data, exporting to 31-35 data, importing from 20-22
csv.reader() method 21

D

data
cleaning up, from outliers 40-46
exporting, to CSV file 31-35
exporting, to JSON file 31-35
exporting, to Microsoft Excel file 31-35 importing, from CSV file 20-22
importing, from database 36-40
importing, from fixed-width datafile 25, 26 importing, from JSON resource 28-30 importing, from Microsoft Excel file 22-25 importing, from tab-delimited file 27, 28 plotting on map, Basemap used 172-176 plotting on map, Google Map API used

177-183
database
data, importing from 36-40

data table
adding, to figure 116-118
data visualization, types
bar charts 72
histograms 72
line graphs 72
pie charts 72
deactivate command 9
dialect 27
Distutis 8
Django 1.1 8
drawMap() function 182
dump() method 35
DVI to PNG converter 247

E

Enthought Python Distribution (EPD) 10 equential colormap 205
error bars

about 99, 237
drawing 99-101

making 237-239

F
figtext function 241
figure
data table, adding 116-118
figure() function 76, 89
file
reading 46-48

filesystem tree
visualizing, polar bar used 134-137
fill_between() function 104, 128
filled areas
plotting 103, 104
fill() function 105
fixed-width datafile
data, importing from 25, 26
font properties
family 242
fontproperties 243
fontsize 242
fontstretch 243
fontstyle 242
fontweight 242
size 242
stretch 243
style 242
using 240-246
variant 242
weight 242
format_data() function 167
freetype 6
freetype 1.4+ 7
FuncAnimation (TimedAnimation) 147

G

Gantt chart

about 232
making 232-237

get_captcha method 187
get() method 30
get_size function 137
Ghost script 247
GitHub

URL 29
Glumpy 151
Glumpy Quickstart

used, for animating with OpenGL 155 Google Data Visualization Library 177
Google Developer

URL 183
Google Geochart 178
Google Map API

used, for plotting data on map 177-183 Google Visualization API 177
grids

about 121-123
customizing 121-124
setting 89, 90

GTK Tools 139
H

hist() function 98
histograms
about 97
making 97-99
using 210-217
hold property 74
Homebrew project 10
horizontalalignment property 243 HTTP Protocol and Response message
URL 183

I
ImageChops module 159
image data
importing, into NumPy arrays 50-56

ImageDraw module 158
ImageFilter module 159
Image module
im.crop(box) method 158
im.filter(filter) method 158
im.histogram() method 158
im = Image.open(filename) method 158 im.resize(size, filter) method 158
im.rotate(angle, filter) method 158 im.split() method 158
im.transform(size, method, data, filter) method 158
image processing
example 56
with PIL 158-163
image processing, Python 50-56 images
displaying, with other plots 168-172 plotting 164-168
imread function 168
init() function 149
installation, matplotlib
steps 6-8
on Mac OS X 10, 11
on Windows 11, 12
installation, NumPy 6-8
installation, PIL
for image processing 13
installation, requests module 14 installation, SciPy 6-8
installation, SQLite library 36 installation, virtualenv 8, 9
installation, virtualenvwrapper 8, 9 intsallation, Python 11, 12
IPython 8, 153
IPython Notebook 255
isolines 125

J
JavaScript Object Notation (JSON) 28

JSON file

data, exporting to 31-35
json.loads() function 31
JSON resource
data, importing from 28-30

L

labels
setting 89, 90
LaTeX
about 246, 247
syntax 246
used, for rendering text 246-250
LaTex syntax 81
LaTeX system 247
legend
adding 92-94
legend() function 93
libpng 6
libpng 1.2 7
line markers
list 87
line plot
about 72-78
format strings, defining 84-86 properties 86
properties, defining 84-86 styles, defining 84-86
linestyles
list 87
linspace function 64
load_data function 167
load_files() function 163
loadtxt() function 21
logarithmic plot
about 190-193
rules 190

M

map

data plotting, Basemap used 172-176 data plotting, Google Map API used 177-183

matplotlib
about 72
animation 146-149
installing 6-8
installing, on Mac OS X 10, 11
installing, on Windows 11, 12
plot, elements 73

matplotlib API 251
matplotlib parameters
customizing, in code 14, 16
customizing, per project 16, 17
matplotlib.pylab interface 251
matplotlibrc configuration file
location 17
settings 17
matplotlib software
backends 251
matplotlib API 251
matplotlib.pylab interface 251
Mayavi 151
Median absolute deviation (MAD) 40 Median Filter 69
meshgrid() function 228
meshgrid property 204
Microsoft Excel file
data, exporting to 31-35
data, importing from 22-25 mkvirtualenv ENV command 9 mplot3d 139
multialignment property 243

N

Natgrid 139
NetCDF 177
next() function 47
noise signal

smoothing, in real-world data 64-70

NumPy
about 6
installing 6-8
URL 6

NumPy arrays
image data, importing into 50-56
O

object-oriented API (OO API)
about 250
differentiating, with pyplot 250-255

open() function 163

OpenGL
about 151
animating with 150-154
animating with, Glumpy Quickstart used 155 animating with, Pyglet Quickstart used 154

OpenRefine
URL 46
Optical Character Recognition (OCR) 188
outliers
data, cleaning up 40-46

P

pie charts
about 101
making 101, 103

PIL
about 12, 53
installing, for image processing 13
URL 13
used, for image processing 158-163

Pillow
URL 13
plot
background color, defining 89 color, defining 88
elements 73
plot() function 73, 142, 167
plot types
bar plot 72-78
box plot 77
contour plot 125
cosine plot 78-81
defining 72-78
line plot 72-78
logarithmic plot 190
polar plot 132
scatter plots 105
sine plot 78-81
stacked charts 72-78
stem plot 198
stream plot 201
whiskers plot 77
polar() function 132
polar plot
about 132
drawing 131-133
probability distribution 57
Processing
URL 155
proTeX system 247
Pyglet 151
Pyglet Quickstart
used, for animating with OpenGL 154
PyPi
URL 25
pyplot
differentiating, with OO API 250-255
pyplot function 207
Pyprocessing 155
Python

CSV module 20
file, reading 46, 47, 48
image processing 50-56
intsalling 11, 12
Python 2.3 8
Python 2.6 8
Python 2.7+ Version 6
Python 3.3+ Version 6
Python Distribution Utilities 8 Python Imaging Library. *See* PIL

R

rainbow colormap 206
random module 57
read() function 47
read_png() function 167
real-world data

noise signal, smoothing 64-70 reCAPTCHA
URL 188
recaptcha-client
URL 188
requests module
about 29
installing 14
Response.json() method 30 run() function 155

S

save() function 149, 163
scatter() function 107, 142
scatterhist() function 215
scatter plots

about 105
drawing, with colored markers 105-107 using 210-217

scikit-image
URL 56
SciPy

about 6
installing 6-8
SciPy Cookbook
URL 66
SciPy signal toolbox
implementation 69
seek() function 47
seismic colormap 206
setp() function 85
setstate() function 63
shadow
adding, to chart line 113-116
sheets() method 24
show() function 171, 237
sine plot
drawing 78-81
spectrogram
about 193-197
spines
about 95
moving, to center 95, 96
split() function 28
SQLite db file 38
SQLite library
installing 36
stacked chart plot 72-78
standard deviation 57
statistical population 57
stem() function 198
stem plot
about 198
creating 198-201
formatters 198
streaming data source
reading 48, 49
stream plot
about 201
drawing 201-205
used, for visualizing vector field 201

struct module
about 25
URL 25
subplot
using 118-120
suptitle function 241

T

tab-delimited file
data, importing from 27, 28
Table Visualization 178
tell() function 47
terrain colormap 206
test_* function 153
text
alignment properties 243 rendering, with LaTeX 246-250
text function 241
text properties
using 240-246
ticks
about 89
setting 89, 90
TimedAnimation (Animation) class 147
title function 241
toolkit 139
transformations 113

U
Ubuntu 13.04 246
under-plot area
filling 128-131
urllib2 module 14
usevlines() function 218
V

variance 57
verticalalignment property 243 virtualenv

about 8
installing 8, 9
virtualenvwrapper

about 9
installing 8, 9
URL 9

W

whiskers plot
about 77
making 229-232

WIMP
URL 158
windowing algorithm 68
workon ENV command 9
write_data() function 35
WYSIWYG
URL 158

X
xlabel function 241
Y
ylabel function 241
Thank you for buying

Python Data Visualization Cookbook
About Packt Publishing

Packt, pronounced 'packed', published its first book " *Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the

software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cuttingedge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Python Geospatial
Development - Second
Edition

ISBN: 978-1-78216-152-3 Paperback: 508 pages Learn to build sophisticated mapping applications from scratch using Python tools for geospatial development 1. Build your own complete and sophisticated mapping applications in Python

2. Walks you through the process of building your own online system for viewing and editing geospatial data

3. Practical, hands-on tutorial that teaches you all about geospatial development in Python
Programming ArcGIS 10.1 with Python Cookbook
ISBN: 978-1-84969-444-5 Paperback: 304 pages

Over 75 recipes to help you automates geoprocessing tasks, create solutions, and solve problem for ARcGIS with Python

1. Learn how to create geoprocessing scripts with ArcPy
2. Customize and modify ArcGIS with Python 3. Create time-saving tools and scripts for ArcGIS

NumPy Cookbook

ISBN: 978-1-84951-892-5 Paperback: 226 pages Over 70 interesting recipes for learning the Python open source mathematical library, NumPy

1. Do high performance calculations with clean and efficient NumPy code
2. Analyze large sets of data with statistical functions 3. Execute complex linear algebra and mathematical computations

NumPy Beginner's Guide Second Edition

ISBN: 978-1-78216-608-5 Paperback: 310 pages

An action packed guide using real world examples of the easy to use, high performance, free open source NumPy mathematical library

1. Perform high performance calculations with clean and efficient NumPy code
2. Analyze large datasets with statistical functions 3. Execute complex linear algebra and
mathematical computations