Katie Cunningham

Sams Teach Yourself

Python

in 24 Hours

SAMS

FREE SAMPLE CHAPTER

Katie Cunningham

Sams Teach Yourself Python in 24 Hours

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself Python in 24 Hours

ISBN-13: 978-0-672-33687-4
ISBN-10: 0-672-33687-1

Library of Congress Control Number: 2013944085

Printed in the United States of America

First Printing October 2013

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

Editor-in-Chief
Mark Taub

Executive Editor
Debra Williams Cauley

Development Editor
Michael Thurston

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Bart Reed

Indexer
Lisa Stumpf

Proofreader
Dan Knott

Technical Editors
Doug Hellmann
Gabriel Nilsson

Publishing Coordinator
Kim Boedigheimer

Cover Designer
Mark Shirar

Senior Compositor
Gloria Schurick

Contents at a Glance

	Preface	xiii
	Introduction	1
HOUR 1	Installing and Running Python	5
HOUR 2	Putting Numbers to Work in Python	17
HOUR 3	Logic in Programming	27
HOUR 4	Storing Text in Strings	37
HOUR 5	Processing Input and Output	49
HOUR 6	Grouping Items in Lists	61
HOUR 7	Using Loops to Repeat Code	71
HOUR 8	Using Functions to Create Reusable Code	81
HOUR 9	Using Dictionaries to Pair Keys with Values	95
HOUR 10	Making Objects	103
HOUR 11	Making Classes	113
HOUR 12	Expanding Classes to Add Functionality	125
HOUR 13	Using Python's Modules to Add Functionality	139
HOUR 14	Splitting Up a Program	149
HOUR 15	Providing Documentation for Code	159
HOUR 16	Working with Program Files	171
HOUR 17	Sharing Information with JSON	183
HOUR 18	Storing Information in Databases	197
HOUR 19	Using SQL to Get More out of Databases	209
HOUR 20	Developing for the Web with Flask	223
HOUR 21	Making Games with PyGame	241
HOUR 22	Saving Your Code Properly Through Versioning	259
HOUR 23	Fixing Problem Code	273
HOUR 24	Taking the Next Steps with Python	285
	Index	295

Table of Contents

Preface xiii

Who This Book Is For For ... xiii

How This Book Is Organized ... xiii

Introduction 1

Learning to Program ... 1

Why Python? ... 2

Getting Started ... 2

How This Book Works ... 3

What to Do If You Get Stuck ... 3

HOUR 1 Installing and Running Python 5

Discovering Your Operating System ... 5

Setting Up Python on Windows ... 7

Setting Up Python on a Mac ... 11

Summary ... 15

Q&A ... 15

Workshop ... 16

HOUR 2 Putting Numbers to Work in Python 17

Storing Information with Variables ... 17

Doing Math in Python ... 20

Comparing Numbers ... 23

Applying Python Math in the Real World ... 24

Summary ... 25

Q&A ... 26

Workshop ... 26

HOUR 3 Logic in Programming 27

Using a Basic `if` Statement ... 27

Creating Blocks ... 28

Adding an `else` to an `if` ... 29

Testing Many Things with `elif` 30
True and False Variables 31
Using `try/except` to Avoid Errors 32
Applying Logic to Real-World Problems 34
Summary 35
Q&A 35
Workshop 36

HOUR 4 Storing Text in Strings 37
Creating Strings 37
Printing Strings 38
Getting Information About a String 38
Math and Comparison 40
Formatting Strings 42
Using Strings in the Real World 46
Summary 47
Q&A 47
Workshop 48

HOUR 5 Processing Input and Output 49
Getting Information from the Command Line 49
Getting a Password 53
Cleaning Up User Input 54
Formatting Output 55
Managing Input and Output in the Real World 57
Summary 58
Q&A 58
Workshop 58

HOUR 6 Grouping Items in Lists 61
Creating a List 61
Getting Information About a List 63
Manipulating Lists 64
Using Math in Lists 65
Ordering Lists 66
Comparing Lists 67

Using Lists in the Real World ... 67
Summary ... 68
Q&A ... 68
Workshop ... 69

HOUR 7 Using Loops to Repeat Code **71**
Repeating a Set Number of Times ... 71
Repeating Only When True ... 76
Using Loops in the Real World ... 77
Summary ... 79
Q&A ... 79
Workshop ... 80

HOUR 8 Using Functions to Create Reusable Code **81**
Creating a Basic Function ... 81
Passing Values to Functions ... 82
Variables in Functions: Scope ... 86
Grouping Functions Within a Function ... 88
Sending a Varying Number of Parameters ... 88
Using Functions in the Real World ... 89
Summary ... 92
Q&A ... 92
Workshop ... 93

HOUR 9 Using Dictionaries to Pair Keys with Values **95**
Creating a Dictionary ... 95
Getting Information About a Dictionary ... 97
Comparing Dictionaries ... 98
Using Dictionaries in the Real World ... 99
Summary ... 101
Q&A ... 101
Workshop ... 101

HOUR 10 Making Objects **103**
Object-Oriented Programming ... 103
Planning an Object ... 107

Making Objects Out of Objects 108
Using Objects in the Real World 110
Summary 111
Q&A 111
Workshop 111

HOUR 11 Making Classes 113
Making a Basic Class Statement 113
Adding Methods to Classes 114
Setting Up Class Instances 116
Using Classes in the Real World 119
Summary 122
Q&A 122
Workshop 122

HOUR 12 Expanding Classes to Add Functionality 125
Built-in Extras 125
Class Inheritance 130
When to Expand Classes in the Real World 134
Summary 136
Q&A 136
Workshop 137

HOUR 13 Using Python's Modules to Add Functionality 139
Python Packages 139
Using the `random` Module 140
Using the `datetime` Module 143
Finding More Modules 145
Using Modules in the Real World 146
Summary 147
Q&A 147
Workshop 148

HOUR 14 Splitting Up a Program 149
Why Split Up a Program? 149
Deciding How to Break Up Code 150

How Python Finds a Program's Code 152
Splitting Up Code in the Real World 155
Summary 157
Q&A 157
Workshop 158

HOUR 15 Providing Documentation for Code 159
The Need for Good Documentation 159
Embedding Comments in Code 160
Explaining Code with Docstrings 162
Including README and INSTALL 164
Providing Documentation in the Real World 167
Summary 168
Q&A 168
Workshop 169

HOUR 16 Working with Program Files 171
Reading to and Writing from Files 171
Creating Files 174
Getting Information About a Directory 175
Getting Information About a File 178
Using Files in the Real World 180
Summary 181
Q&A 181
Workshop 181

HOUR 17 Sharing Information with JSON 183
The JSON Format 183
Working with JSON Files 185
Saving Objects as JSON 188
Creating Custom Dictionaries 189
Using JSON in the Real World 191
Summary 194
Q&A 194
Workshop 195

HOUR 18 Storing Information in Databases 197
Why Use Databases? 197
Talking to Databases with SQL 198
Creating a Database 200
Querying the Database 203
Using Databases in the Real World 205
Summary 207
Q&A 207
Workshop 208

HOUR 19 Using SQL to Get More out of Databases 209
Filtering with `WHERE` 210
Sorting with `ORDER BY` 214
Getting Unique Items with `DISTINCT` 215
Updating Records with `UPDATE` 215
Deleting Records with `DELETE` 216
Using SQL in the Real World 217
Summary 220
Q&A 220
Workshop 221

HOUR 20 Developing for the Web with Flask 223
What Is Flask? 223
Installing Flask 225
Making Your First Flask App 228
Adding Templates 231
Using Frameworks in the Real World 237
Summary 238
Q&A 238
Workshop 239

HOUR 21 Making Games with PyGame 241
What Is PyGame? 241
Installing PyGame 242
Creating Screens 243
Creating Shapes 245

Moving Things Around on the Screen 248
Getting Input from the User 250
Drawing Text 252
Using PyGame in the Real World 253
Summary 257
Q&A 257
Workshop 258

HOUR 22 Saving Your Code Properly Through Versioning 259
What Is Versioning? 259
Versioning with Git and GitHub 261
Managing Code in a Repository 263
Experimental Changes with Branches 267
Determining What Not to Push 270
Summary 271
Q&A 271
Workshop 271

HOUR 23 Fixing Problem Code 273
When Your Code Has a Bug 273
Locating Errors with a Traceback 274
Finding Errors with the pdb Debugger 275
Searching the Internet for Solutions 278
Trying a Fix 279
Finding Outside Support 280
Summary 282
Q&A 282
Workshop 283

HOUR 24 Taking the Next Steps with Python 285
Interesting Projects 285
Attending Conferences 288
Working with Linux 288
Contributing to Python 290
Contributing to Other Projects 290

Learning Another Language 290
Looking Forward to Python 3 291
Recommended Reading 292
Recommended Websites 292
Summary 293
Q&A 293
Workshop 293

Index **295**

About the Author

Katie Cunningham is a Python developer at Cox Media Group. She's a fervent advocate for Python, open source software, and teaching people how to program. She's a frequent speaker at open source conferences, such as PyCon and DjangoCon, speaking on beginners' topics such as someone's first site in the cloud and making a site that is accessible to everyone.

She also helps organize PyLadies in the DC area, a program designed to increase diversity in the Python community. She has taught classes for the organization, bringing novices from installation to writing their first app in 48 hours.

Katie is an active blogger at her website (http://therealkatie.net), covering issues such as Python, accessibility, and the trials and tribulations of working from home.

Katie lives in the DC area with her husband and two children.

Dedication

This is dedicated to my family, who helps keep me sane every time
I decide to do this again. Jim, thank you for picking up the slack.
Mom, thank you for taking the kids and offering help every time
I started to look like I was going to fall over.
Kids, thank you for being okay with all the delivery food.

Acknowledgments

This book wouldn't have happened without the help from quite a few people.

First, my editor, Debra Williams Cauley, has been both patient and enthusiastic. Without her, I don't know if I would have ever hit the deadline.

A special thanks goes to my tech editors, Doug Hellmann and Gabriel Nilsson. They were machines when it came to catching my glaring errors, and their suggestions only made this book stronger. Also, a thanks goes out to Richard Jones, who took the time to review my PyGame chapter.

Thanks to Michael Thurston, who made me sound fabulous. I swear, one of these days, I'll learn to spell "installation" right.

Finally, a thank you goes out to the Python community, who has been on hand every time I had a question, needed a sanity check, or just needed some inspiration. You guys are my home.

Preface

Why Python?

I get this question quite a bit. Why should someone learning to program learn Python? Why not a language that was made for beginners, such as Scratch? Why not learn Java or C++, which most colleges seem to be using?

Personally, I believe that Python is an ideal language for beginners. It runs on multiple systems. The syntax (the grammar of the language) isn't fussy. It's easy to read, and many people can walk through a simple script and understand what it's doing without ever having written a single line of code.

It's also ideal because it's easy for a beginner to move on to more advanced projects. Python is used in a number of areas, from scientific computing to game development. A new programmer can almost always find one, if not multiple, projects to fit their tastes.

Who This Book Is For

This book is for those who have never programmed before and for those who have programmed some but now want to learn Python. This is not a book for those who are already experienced developers.

It is assumed you have a computer you have admin rights to. You'll need to install Python, as well as multiple libraries and applications later in the book. The computer does not need to be terribly powerful.

You should also have an Internet connection in order to access some of the resources.

How This Book Is Organized

This book covers the basics of programming in Python as well as some advanced concepts such as object-oriented programming.

- The Introduction and Hour 1 cover the background of Python and installation.
- Hours 2–7 cover some basics of programming, such as variables, math, strings, and getting input.

- Hours 8–12 cover advanced topics. Functions, dictionaries, and object-oriented programming will be discussed.
- Hours 13–15 discuss using libraries and modules, as well as creating your own module.
- Hours 16–19 cover working with data, such as saving to files, using standard formats, and using databases.
- Hours 20 and 21 give a taste of some projects outside of the standard library. In these hours, you will explore creating dynamic websites and making games. These hours are not meant to be complete lessons, but serve instead as a starting point for learning more.
- Hours 22 and 23 go over how to save your code properly, and how to find answers when something has gone wrong.
- Hour 24 goes over what projects you can get involved with, what resources can help you learn more, and how to get more involved in the Python community.

We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: consumer@samspublishing.com

Mail: Sams Publishing
ATTN: Reader Feedback
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

Many people idly contemplate learning how to code. It seems like something that could be of use, but many are too intimidated to jump in and try. Maybe they believe it's too late to start learning a skill like programming, or they believe they don't have enough time. Maybe they get lost too quickly, because the book they found is written for someone with previous experience with coding. It seems like an impossible task. The goal of this book is to break down the concepts behind programming into bite-sized chunks that are easy to digest as well as immediately useful.

Learning to Program

For many people, learning to program seems like an impossible task. It's painted as a field that requires a crazy amount of math, years of education and training, and, once you're done with that, endless hours of constantly banging away at a keyboard.

The truth is, although becoming a full-time developer can take quite a bit of dedication, learning how to write code can be easy. As more of our life touches computers, learning to write code to control them can enhance any career, no matter how nontechnical it may seem. An elementary school teacher might make a website to help students learn their vocabulary. An accountant could automate calculations that normally have to be done by hand. A parent could create a home inventory system to help with generating grocery lists. Nearly every profession and hobby can be enhanced through learning to program.

To put it simply, computers are stupid. Without human input, they don't know what to do. Code is a set of instructions that tells the computer not only what to do, but how to do it. Everything on your computer, from the largest applications (such as Word and video games) to the smallest (such as a calculator), is based on code.

Most code on your computer will be compiled already as an .exe or .app file. For the exercises in this book, we'll either be running them from a file or using the interpreter (which we'll get to in Hour 1, "Installing and Running Python").

Why Python?

Python is a language that is lauded for its readability, its lack of fussiness, and how easy it is to teach. Also, unlike some languages that are created specifically for teaching, it's used in countless places outside of the classroom. People have used Python to write everything from websites to tools for scientific work, from simple scripts to video games. The following is a non-exhaustive list of programs written in Python:

- **YouTube**—A popular site for viewing and sharing videos.
- **The Onion**—A parody news site.
- **Eve Online**—A video game set in space.
- **The Washington Post**—The website runs off of Django, a framework written in Python.
- **Paint Shop Pro**—An image-editing software package.
- **Google**—A significant number of applications at Google use Python.
- **Civilization IV**—A turn-based simulation game.

Python may appear simple, but it's incredibly powerful.

Getting Started

Before we get started, let's go over a list of some things you're going to need. You absolutely must have all these things before you can start learning Python. Here's what you will need:

- **Admin access**—Python doesn't require a very powerful computer to run, but you will need a computer that you have permission to install things on.
- **Internet access**—We're going to be downloading installers, and, later on, talking to web services. It doesn't need to be a fast connection, because many of the items we'll be downloading are rather small.
- **A computer**—It doesn't need to be brand new, but the faster your computer is, the faster your code should run. A computer built in the past five years should be fine.
- **Space**—A dedicated workspace can greatly enhance your ability to pick up new concepts. It should be free from distractions, such as TV.
- **No distractions**—It's almost impossible to learn something new if you have family members interrupting you, phones buzzing, or a TV blaring in the background. A good pair of noise-canceling headphones can be a wonderful asset—if you can't get rid of people and ambient noise.

For most people, the last two items can be the most difficult to get in place, but they're invaluable. Not only will you need them while learning, but you'll need them once you're done with this book and moving on to your own projects. Writing code is a creative endeavor, and requires time and space to do.

How This Book Works

Each chapter is meant to be completed in one hour or less. That includes reading the text and doing the exercises. Ideally, the exercises should be done directly after reading a chapter, so try to set aside time when you not only can focus, but have access to your computer. Not every chapter will require Internet access (those that do will warn you before you dive in).

It may be tempting to dive in to the next chapter after finishing one, but try to give yourself a break. Your brain needs time to integrate the new information, and you need to be rested before diving into more new material.

What to Do If You Get Stuck

There is one thing that applies to every person who writes code: You will get stuck. Sometimes a new concept doesn't seem to be clicking. Sometimes an error won't go away. There are days when everything you touch seems to break.

The key to getting past days like these is to not give in to frustration. Get up, move away from the computer, and go for a walk. Make a cup of tea. Talk to a friend about anything but your misbehaving code. Give yourself a chance to unwind.

When you've given yourself some space from the problem, do a quick self-assessment. Are you tired? A tired developer is a bad developer, no matter how experienced he or she is. Sometimes a bit of coffee helps, but most of the time what you need is some sleep.

If you're not tired, try re-reading the chapter. It might be time to break out the highlighters or take notes. Are some of the terms unfamiliar? Try searching for these terms online.

Is the code not working? Sometimes, you need to delete what you have (or save it in another file) and try again. Later in the book, we'll talk about better ways to debug your code, but rest assured, every developer has had to toss code at some point in his or her life.

This page intentionally left blank

HOUR 1

Installing and Running Python

What You'll Learn in This Hour:

- How to determine what operating system you're running
- How to install and run Python
- How to input basic commands into Python

Installing Python is one of the most important things you'll be doing in this book. Without it, you can't complete the rest of this book! Make sure to take your time in this hour. If you can't pass the exercises at the end of the hour, you'll have problems with every hour after this one.

Discovering Your Operating System

Many people know what kind of computer they have, but have no idea about the specific operating system that's installed on it. Knowing what operating system you're running is vital to learning how to program because it might change what you need to download or how you access certain parts of the system.

In general, if your computer was made by Apple (for example, if it's a MacBook or PowerBook), it's running Mac OS. Most other personal computers are running Windows.

If you ever have issues, you will need to know exactly what version of your OS you're running. On a Mac, click the Apple icon in your menu bar and select About this Mac. A window will pop up with some information about your computer, including the exact version of your OS (see Figure 1.1).

FIGURE 1.1
Finding the exact version of Mac OS.

If you're running a Windows machine, click your Start menu and find the Command Prompt program under Accessories. Clicking it will open the command prompt for your computer (see Figure 1.2).

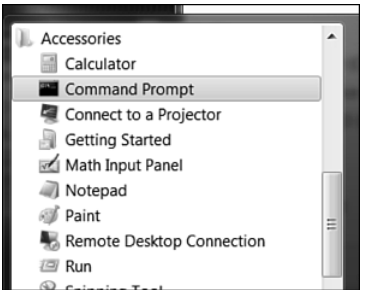

FIGURE 1.2
Finding Command Prompt in Windows.

If you're having trouble finding the command prompt, search for "cmd" in the Start menu's search or run box.

Once Command Prompt is open, type **systeminfo** and press Enter. A bunch of data will print out, but what you need is at the top. Scroll up and look for a line starting with "OS Name." In Figure 1.3, the version is Microsoft Windows 7 Home Premium.

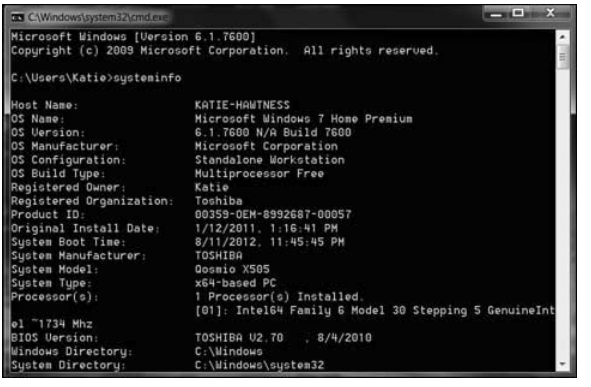

FIGURE 1.3
Finding the Windows version in systeminfo.

Now that you're clear about what operating system you're running, let's install Python and a text editor. We'll cover Windows first. If you're using a Mac, go ahead and skip to the Mac section.

Setting Up Python on Windows

In this section, we'll guide you through installing Python on your Windows machine. Python 2.7 will run on Windows 2000, XP, Vista, and Windows 7 and 8. If your computer was bought after 2000, you're probably running one of these operating systems.

As for memory and hard drive space, Python is designed to run on little memory and take up little space. If you're running any Windows release after XP, you'll be fine.

Installing Python on Windows

Go to http://www.python.org/getit/ in any browser. There, you'll see a list of various downloads for Python. Some of the downloads are for other operating systems, some are the code that makes Python, and some are Python installers made by other companies. We're only interested in the one that will install Python on a Windows machine.

Look for "Python 2.7.5 Windows Installer (Windows binary -- does not include source), as shown in Figure 1.4. The last two numbers (the five and the seven) might change, but you should definitely get the package starting with the two. Python 3 is out, but this book is written for Python 2 (for more information about why we're using version 2 rather than version 3, check the "Q&A" section of this hour). There are some subtle differences between the two that might get confusing down the road if you install the wrong version.

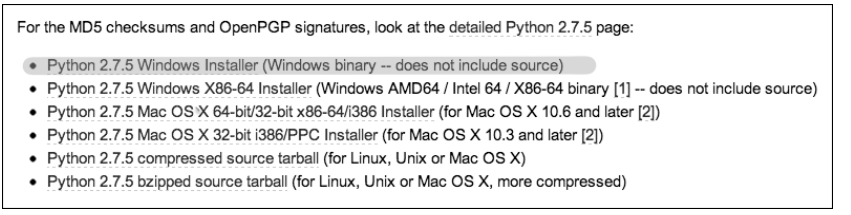

FIGURE 1.4
Windows Installer on Python.org.

Click the link to download the installer. Once it's done, click the downloaded installer to install Python. You should accept most of the default settings. The only one you should consider is whether you want to install Python just for you or for all users. If you're the only person on your computer, the question is moot, but if you share it with others (and they have their own logins), you should decide if you want to install Python for them as well. If you're not sure, install Python for all users because it doesn't significantly change the way their computer works and only adds a few new items to the Start menu.

If you're on Windows Vista or later, you'll likely get a pop-up asking if you want to allow the installer to make changes to your computer. Click Allow (or Okay, or whatever seems to be an affirmative response) to allow the installation to continue.

Once the Python installation is complete, you should have some new items under your Start menu. If you don't have the items in Figure 1.5, try to install Python again. You may have canceled the installation at some point by accident.

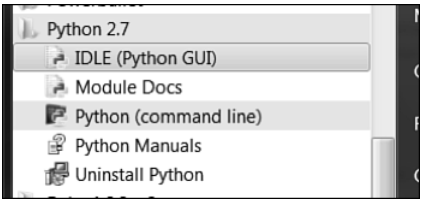

FIGURE 1.5
New Start menu items.

Running Python on Windows

For the book's early hours, we'll be running Python through the Python shell. The Python installer has given us two tools that make getting to the shell pretty easy: a link to the command line and a program called IDLE, shown in Figure 1.6. From here on out, when you're asked to

open a Python shell, open IDLE. Sometimes you'll be asked to run a file. In that case, open the file in IDLE and then either select Run Module under the Run menu or press F5.

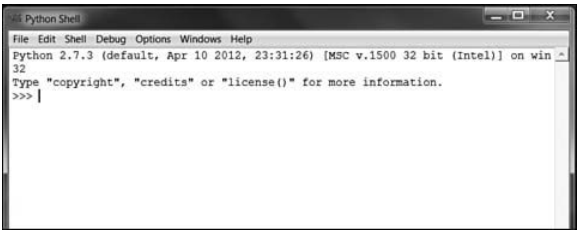

FIGURE 1.6
The Python shell in IDLE.

When you open IDLE, you'll see a screen like the one in Figure 1.6. This is called the Python shell. Here, Python is waiting for you to type in commands, which it will execute right away. Go ahead and enter the following and press Enter:

```
print "Hello, world!"
```

Your screen should look like IDLE in Figure 1.7.

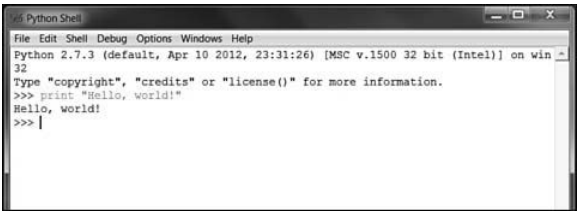

FIGURE 1.7
A line of Python code in IDLE.

Congratulations! You've written your first line of Python code!

Installing a Text Editor on Windows

IDLE comes with a text editor, but you might want one that's a bit more robust to use in the book's later hours. For that, Notepad++ is a good option. It's available for free at http://notepad-plus-plus.org. For the moment, though, IDLE's text editor should be fine.

It's important never to open a Python file with a word processing program such as WordPad or Word. They have a tendency to wreak havoc with formatting and insert items that you may not be able to see. Once they're in there, it can be difficult to find them and remove them.

Getting Around the File System

Though we'll be working with Python through IDLE in the beginning, eventually you'll need to get around your computer via the terminal.

Open a command prompt (this is the window you opened earlier to get information about your system). You should see something like this:

```
C:\Users\YourName\> _
```

Where your cursor is currently blinking is called your command line (though you'll often see it referred to as your "prompt"). The text can be customized, but on most Windows computers, it's set to your current directory (another word for "folder").

To see what your current directory is, use the `cd` command:

```
C:\Users\YourName\> cd
C:\Users\YourName\
```

If you want to move to another directory, add that directory after the `cd` command:

```
C:\Users\YourName\> cd Downloads
C:\Users\YourName\Downloads\>
```

You can also use the full path of the directory you want to move to (that's a line that contains every nested directory):

```
C:\Users\YourName\> cd c:\Users\YourKid\
C:\Users\YourKid\>
```

You can also get a list of everything in a directory by using the `dir` command. If you use the command on its own, it will give you a list of the files in your current directory. If you give the command a directory, it will return all the contents of that directory.

```
C:\Users\YourName\projects> dir
c:\Users\YourName\projects> dir
 Volume in drive C is TI105970W0D
 Volume Serial Number is 52G3-1C5A

 Directory of c:\Users\YourName\projects

12/08/2012  09:38 AM    <DIR>          .
12/08/2012  09:38 AM    <DIR>          ..
12/08/2012  09:36 AM    <DIR>          rogue
06/20/2012  02:24 PM               198 todo.txt
12/08/2012  09:36 AM    <DIR>          website
               1 File(s)            198 bytes
               4 Dir(s)  79,784,599,552 bytes free
```

Each line tells you the following:

- When the file was created
- Whether it's a directory (indicated by `<DIR>`)
- How big the file is
- What the directory or file is called

If you want to make a new directory, use the `mkdir` command. This command requires that you tell it what you want to name the directory. If you just use the command on its own, you'll get an error.

```
C:\Users\YourName\projects> mkdir python
C:\Users\YourName\projects\> dir
Volume in drive C is TI105970W0D
 Volume Serial Number is 52G3-1C5A

 Directory of c:\Users\Katie\projects

12/08/2012  09:40 AM    <DIR>          .
12/08/2012  09:40 AM    <DIR>          ..
12/08/2012  09:40 AM    <DIR>          python
12/08/2012  09:36 AM    <DIR>          rogue
06/20/2012  02:24 PM               198 todo.txt
12/08/2012  09:36 AM    <DIR>          website
               1 File(s)            198 bytes
5 Dir(s)  79,784,603,648 bytes free
```

Now that you know how to get around on your computer through the command prompt, feel free to move to the "Try It Yourself" section.

Setting Up Python on a Mac

In this section, we will go over setting up Python on your Mac and installing a text editor. If you're using a Windows machine, feel free to skip this section.

Installing Python on a Mac

If you're running a Mac, you already have Python installed! There's no need to download anything extra. Though there are some slight differences between the types of Python on older Macs, those differences shouldn't affect the activities we'll be doing in this book.

Running Python on a Mac

Whenever you're asked to run the Python shell, you'll need to start up IDLE. Sometimes, you'll be asked to run a file. In that case, start up IDLE and open the file (look under the File menu). Once the file is open, make sure to select the window with the code you want to run and then select Run Module under the Run menu.

In order to get IDLE running, you'll need to open up a terminal window. Click the search icon in your toolbar and search for "terminal." You should see something like the screen shown in Figure 1.8.

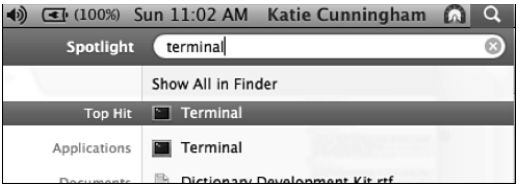

FIGURE 1.8
Finding the terminal.

Clicking Terminal will bring up a terminal window like the one shown in Figure 1.9. A terminal window gives you access to your computer through the command line.

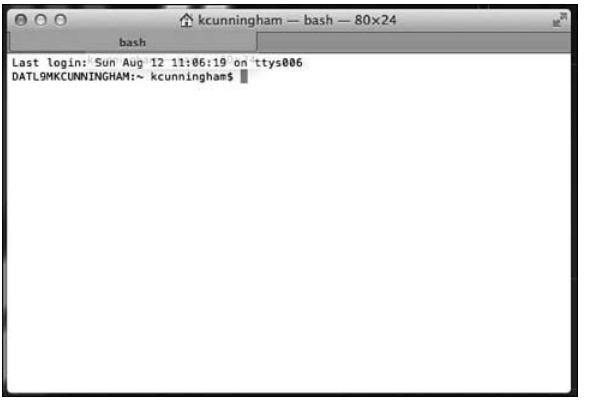

FIGURE 1.9
The terminal window.

We'll go over some of the things you can do in the terminal later. For now, let's start up IDLE. On the command line, type **idle** and press Return. A new program will start up that looks like the screen in Figure 1.10. This is the Python shell. Python is actively running and waiting for you to input commands.

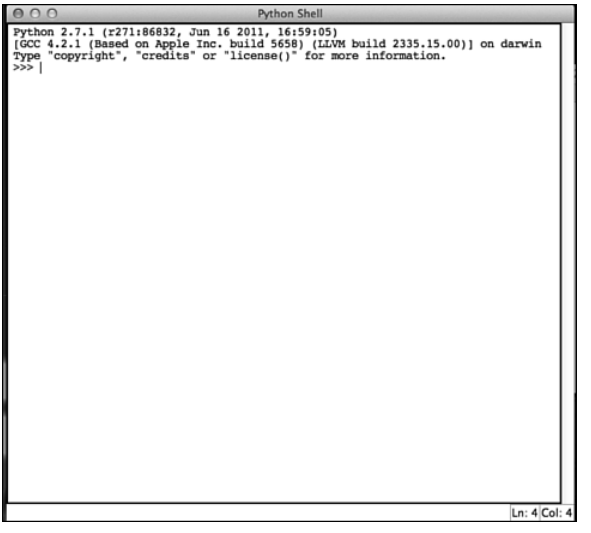

FIGURE 1.10
The Python shell in IDLE.

Go ahead and type **print "Hello, world!"** and press Return. You should see something like Figure 1.11.

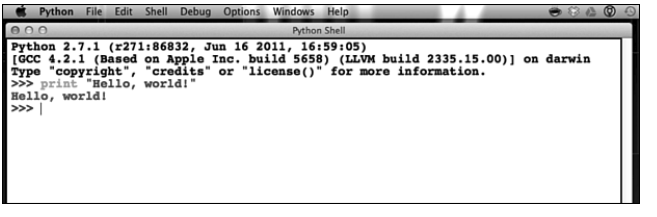

FIGURE 1.11
A line of Python code in IDLE.

Congratulations! You've written your first line of Python code!

By default, the font is set a bit small for some monitors. If you want to change the font size, go to Configure IDLE under the Options menu. There, you can make the font as big as you need.

Installing a Text Editor on a Mac

In the book's early hours, you'll be working with the shell. As your programs grow, however, you'll need a text editor that is geared toward writing code. A great free text editor is

TextWrangler, found at http://www.barebones.com/products/textwrangler/download.html. Download the disk image (that's the installer), and once it's done downloading, click it to install.

It's very important that you do not open your code in any word processing program such as Word or TextEdit. Programs like that can reformat your code and insert items you can't see. In the best case, your code will look ugly. In the worst, and most common case, your code will simply refuse to run.

Getting Around the File System

Though we'll be working with Python through IDLE in the beginning, eventually you'll need to get around your computer via the terminal.

Open a terminal window. You should see something like this:

```
ComputerName: ~$ _
```

Where your cursor is currently blinking is called your command line (though you'll often see it referred to as your "prompt"). The text can be customized, but on most Macs, it's set to your computer's name and your current directory (another word for "folder"). The tilde (~) is a shortcut for your home directory, which is often /Users/Yourusername/.

To see what your current directory is, use the `pwd` command:

```
ComputerName: ~$  pwd
/Users/YourName/
```

If you want to move to another directory, use the `cd` command:

```
ComputerName: ~$ cd Desktop
ComputerName: ~/Desktop/$
```

You can also use the full path of the directory you want to move to (that's a line that contains every nested directory):

```
ComputerName: ~$ cd /Users/YourKid/
ComputerName: /Users/YourKid$
```

Note that you'll often hear people call directories "folders." These are synonyms, and are often used interchangeably.

You can also get a list of everything in a directory by using the `ls` command. If you use the command on its own, it will give you a list of the files in your current directory. If you give it a directory, it will return all the contents of that directory.

```
ComputerName: ~$ ls
ariel.pubkey.asc        Documents               server-misc
asn1                    ldap                    sh-lost
bin                     linux                   signatures
ca-admin                logs                    sounds

ComputerName: ~$ ls Documents
homework1.doc
resume.doc
todo.txt
```

If you want to make a new directory, use the `mkdir` command. This command requires that you give it some sort of value, so it knows what to call the directory.

```
ComputerName: ~$  mkdir projects
ComputerName: ~$ ls[ADD]
```

TRY IT YOURSELF ▼

Testing Your Python Installation

To determine if everything is running correctly, start the Python shell and type in the following commands. The output should match what appears in the right column.

Commands	Output
`print "Hello, world!"`	`Hello, world!`
`5 + 1`	`6`
`import random`	`>>>` (Nothing should appear to happen.)
`random.random()`	(A long decimal number, such as 0.33493820948329084203)

Summary

In this hour, we installed Python on your machine, installed a text editor, and tried out a few Python commands. You also learned what operating system you're running, and you learned some basics about how to move around in the file system.

Q&A

Q. Why am I getting Python 2.7 rather than Python 3?

A. Whenever a new version of Python comes out, it takes a while for everyone who has written libraries that use Python to catch up. Python 3 is great, but some of the libraries we'll be using later haven't moved over to it yet.

Q. Will I have to start learning all over again when everyone moves to Python 3?

A. Not at all! Much of the functionality from Python 2 has been moved over to Python 3. The vast majority of Python 3 will be very familiar to you, once you've completed all 24 hour lessons in this book. Once you feel ready to look into Python 3, check out the guide on Python.org. It will catch you up on all the changes that have been implemented.

Q. Are there any operating systems besides Mac and Windows?

A. There are! Linux is a popular operating system among Python developers. Linux comes in many flavors—from those designed for enormous enterprise systems to those designed for schools and children. These are called "distributions." With the exception of a few made for large businesses, all Linux distributions are free. Some, such as Ubuntu, even have an installer that allows Windows users to dual boot: A user can start up his or her machine in Ubuntu or Windows. For more on Linux, check out the "Next Steps" section at the end of this book.

Workshop

The Workshop contains quiz questions and exercises to help you solidify your understanding of the material covered. Try to answer all questions before looking at the answers that follow.

Quiz

1. What version of Python am I running?
2. What is another word for "folder"?
3. True or false? If I want to edit code in a text editor, I should use Microsoft Word or WordPad.

Answers

1. Python 2.7. There might be another number after the 7 (for example, 2.7.5), but that number can be ignored.
2. Directory. This is how we'll be referring to folders in this book.
3. False! Rich editors such as Word and WordPad will wreak havoc on your code! Use a code editor such as Notepad++ for Windows or TextWrangler on the Mac.

Exercises

1. Through the command line, create a new folder called "projects" in your home folder. Then, change into that directory and create another folder called "python".
2. In your text editor, create a new file called `hello.py` in your new python directory (the one you made in Exercise 1). Type `print "Hello, world"` into it and then save and close it. In your command line, see how big the file is.

HOUR 4

Storing Text in Strings

What You'll Learn in This Hour:

- How to create and print strings
- How to get information about stored text
- How to use math with stored text
- How to format strings
- When to use strings in the real world

When Python wants to store text in a variable, it creates a variable called a *string*. A string's sole purpose is to hold text for the program. It can hold anything—from nothing at all ('') to enough to fill up all the memory on your computer.

Creating Strings

Creating a string in Python is very similar to how we stored numbers in the last hour. One difference, however, is that we need to wrap the text we want to use as our string in quotes. Open your Python shell and type in the following:

```
>>> s = "Hello, world"
>>> s
'Hello, world'
```

The quotes can be either single (') or double ("). Keep in mind, though, that if you start with a double quote, you need to end with a double quote (and the same goes for single quotes). Mixing them up only confuses Python, and your program will refuse to run. Look at the following code, where the text "Harold" starts with a double quote but ends with a single quote:

```
>>> name = "Harold'
File "<stdin>", line 1
name = "Harold'
^ SyntaxError: EOL while scanning string literal
```

As you can see, we got an error. We have to make the quote types match:

```
>>> name = "Harold"
>>> name
'Harold'
>>> name2 = 'Harold'
'Harold'
```

Printing Strings

In the examples so far, Python prints out strings with the quotes still around them. If you want to get rid of these quotes, use a `print` statement:

```
>>> greeting = "Hello"
>>> print greeting
Hello
```

A `print` statement usually prints out the string, then moves to the next line. What if you don't want to move to the next line? In this case, you can add a comma (,) to the end of the `print` statement. This signals Python not to move to a new line yet. This only works in a file, though, because the shell will always move to the next line.

In this example, we print out an item along with the price on the same line:

```
print 'Apple: ',
print '$ 1.99 / lb'
```

When we run it, we get this:

```
Apple:  $ 1.99 / lb
```

We can even do calculations between the two `print` statements, if we need to. Python will not move to a new line until we tell it to.

Getting Information About a String

In Hour 2, "Putting Numbers to Work in Python," variables were compared to cups because they can hold a number of things. Cups themselves have some basic functions, too, whether they contain something or not. You can move them around, you can touch their side to see if what's in them is hot or cold, and you can even look inside them to see if there's anything in there. The same goes with strings.

Python comes with a number of built-ins that are useful for getting information about the stored text and changing how it's formatted. For example, we can use `len()` to see how long a string is.

In the following example, we want to see how long a name is:

```
>>> name = "katie"
>>> len(name)
5
```

In this case, the length of the string held in `name` is five.

In Python, variables also come with some extra capabilities that allow us to find out some basic information about what they happen to be storing. We call these methods. Methods are tacked on to the end of a variable name and are followed by parentheses. The parentheses hold any information the method might need. Many times, we leave the parentheses blank because the method already has all the information it requires.

One set of methods that comes with strings is used to change how the letters are formatted. Strings can be converted to all caps, all lowercase, initial capped (where the first letter of the string is capitalized), or title case (where the first letter and every letter after a space is capitalized). These methods are detailed in Table 4.1.

TABLE 4.1 String-Formatting Methods

Method	Description	Example
`.upper()`	Converts all letters to uppercase (a.k.a. all caps).	'HELLO WORLD'
`.lower()`	Converts all letters to lowercase.	'hello world'
`.capitalize()`	Converts the first letter in a string to uppercase and converts the rest of the letters to lowercase.	'Hello world'
`.title()`	Converts the first letter, and every letter after a space or punctuation, to uppercase. The other letters are converted to lowercase.	'Hello World'

These methods are appended to the end of a string (or variable containing a string):

```
>>> title = "wind in the willows"
>>> title.upper()
'WIND IN THE WILLOWS'
>>> title.lower()
'wind in the willows'
>>> title.capitalize()
'Wind in the willows'
>>> title.title()
'Wind In The Willows'
```

These methods are nondestructive. They don't change what's stored in the variable. In the following example, note that the string stored in `movie_title` isn't changed, even though we used `.upper()` on it:

```
>>> movie_title = "the mousetrap"
>>> movie_title.upper()
'THE MOUSETRAP'
>>> movie_title '
the mousetrap'
```

We can also see if certain things are true about a string. `is_alpha()` and `is_digit()` are two popular methods, especially when checking to see if a user put in the correct type of data for a string.

In the following string, we check to see that `birth_year` is composed of all digits and that `state` is nothing but letters:

```
>>> birth_year = "1980"
>>> state = "VA"
>>> birth_year.isdigit()
True
>>> state.isalpha()
True
```

Had `birth_year` contained any letters or symbols (or even spaces), `isdigit()` would have returned `False`. With `state`, had it contained any numbers or symbols, we would have gotten `False` as well.

```
>>> state = "VA"
>>> state.isdigit()
False
```

Math and Comparison

Just as with numbers, you can perform certain kinds of math on strings as well as compare them. Not every operator works, though, and some of the operators don't work as you might expect.

Adding Strings Together

Strings can also be added together to create new strings. Python will simply make a new string out of the smaller strings, appending one after the next.

In the following example, we take the strings stored in two variables (in this case, someone's first name and last name) and print them out together:

```
>>> first_name = "Jacob"
>>> last_name = "Fulton"
>>> first_name + last_name
'JacobFulton'
```

Note that Python doesn't add any space between the two strings. One way to add spaces to strings is to add them explicitly to the expression.

Let's add a space between the user's first and last names:

```
>>> first_name + " " + last_name
'Jacob Fulton'
```

Multiplication

You can do some funny things with multiplication and strings. When you multiply a string by an integer, Python returns a new string. This new string is the original string, repeated X number of times (where X is the value of the integer).

In the following example, we're going to multiply the string 'hello' by a few integers. Take note of the results.

```
>>> s = 'hello '
>>> s * 5
'hello hello hello hello hello'
>>> s * 10
'hello hello hello hello hello hello hello hello hello hello '
>>> s * 0
''
```

What happens if we store an integer in a string?

```
>>> s = '5'
>>> s * 5
55555
```

Normally, if we multiplied 5 by 5, Python would give us 25. In this case, however, `'5'` is stored as a string, so it's treated as a string and repeated five times.

There's some limitations to string multiplication, however. Multiplying by a negative number gives an empty string.

```
>>> s = "hello"
>>> s * -5
''
```

Multiplying by a `float` gives an error:

```
>>> s * 1.0
Traceback (most recent call last):
File "<stdin>", line 1, in <module> TypeError: can't multiply sequence by
non-int of type 'float'
```

Comparing Strings

It's possible to compare strings just as you would numbers. Keep in mind, however, that Python is picky about strings being equal to each other. If the two strings differ, even slightly, they're not considered the same. Consider the following example:

```
>>> a = "Virginia"
>>> b = "virginia"
>>> a == b
False
```

Although `a` and `b` are very similar, one is capitalized and one isn't. Because they aren't exactly alike, Python returns `False` when we ask whether they are alike.

Whitespace matters, too. Consider the following code snippet:

```
>>> greet1 = "Hello "
>>> greet2 = "Hello"
>>> greet1 == greet2
False
```

`greet1` has a space at the end of its string whereas `greet2` does not. Python looks at whitespace when comparing strings, so the two aren't considered equal.

Operators That Don't Work with Strings

In Python, the only operators that work with strings are addition and multiplication. You can't use strings if you're subtracting or dividing. If you try this, Python will throw an error and your program will stop running.

```
>>> s = "5"
>>> s / 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

If you ever see an error like this one (unsupported operand type), it usually means that the data type you're trying to use doesn't know how to use that operator.

Formatting Strings

There are many ways to format strings—from removing extra spaces to forcing new lines. You can also add in tabs as well as search and replace specified text.

Controlling Spacing with Escapes

Until now, we've been printing strings out on one line. What if we need to print out something on multiple lines? We can use the special combination of a backslash and "n" (`\n`). Every time we insert this into a string, Python will start printing on the next line.

```
>>> rhyme = "Little Miss Muffett\nSat on a tuffet\nEating her curds and whey."
>>> print rhyme
Little Miss Muffett
Sat on a tuffet
Eating her curds and whey.
```

The backslash is a special character in strings. It's called an escape, and it clues Python into the fact that you have some special formatting in mind. You can also use an escape to put a string onto several lines in your code so it's easier to read. The preceding string isn't so easy to read as it is, but we can fix that as follows:

```
>>> rhyme = "Little Miss Muffett\n\
... Sat on a Tuffet\n\
... Eating her curds and whey."
>>> print rhyme
Little Miss Muffett
Sat on a Tuffet
Eating her curds and whey.
```

A new line isn't the only thing you can do with an escape, though. You can also insert tabs with `\t`.

Take note of the spacing in the following example. Each `\t` is replaced with tab when the string is printed.

```
>>> header = "Dish\tPrice\tType"
>>> print header
Dish    Price   Type
```

The escape is also useful for when you have quotes in a string. If you're creating a string that has quotes in it, this can cause some confusion for Python. "Escaping" them lets Python know that you're not done with the string quite yet.

In the following example, the name has a single quote in it. If we don't escape it, Python gives us an error. If we do, however, Python has no problem storing the string.

```
>>> name = 'Harry O'Conner'
File "<stdin>", line 1
name = 'Harry O'Conner'
               ^ SyntaxError: invalid syntax
>>> name = 'Harry O\'Conner'
>>> print name
Harry O'Conner
```

NOTE

Another Way to Deal with Single Quotes

If you don't want to use an escape, you can use double quotes if your string contains single quotes, or vice versa. So, Python will have no issues saving "Harry O'Conner" or 'He said, "Hello" as he opened the door.'

But what if you need to use a backslash in a string? Simple: Just escape the backslash. In other words, if you want to display one backslash, you'll need to enter two backslashes.

In the following example, we want to save a path for a Windows machine. These always include backslashes, so we need to escape the backslash. When we print it, only one backslash appears.

```
>>> path = "C:\\Applications\\"
>>> print path
C:\Applications\
```

Removing Whitespace

Sometimes, a user might put extra whitespace when typing in something for your program. This can be annoying when trying to print out several strings on one line, and it can be downright disastrous if you're trying to compare strings.

In the following example, extra whitespace makes printing out a name difficult. It looks like there's too much space between the first name and middle name. To make matters more difficult, the extra whitespace means that the comparison `first_name == "Hannah"` fails.

```
>>> first_name = "Hannah "
>>> middle_name = "Marie"
>>> print first_name + " " + middle_name
Hannah  Marie
>>> if first_name == "Hannah":
...   print "Hi, Hannah!"
... else:
...   print "Who are you?"
...
Who are you?
```

Strings come with a method, `strip()`, that allows you to strip out all the whitespace at the beginning and end of a string. In the following code snippet, the name Hannah has an extra space tacked onto the end. Using `strip()` removes that space.

```
>>> first_name = "Hannah "
>>> first_name.strip()
'Hannah'
```

`strip()` not only removes all whitespace from around a string, it can remove other characters you specify. This time, Hannah is surrounded by a number of asterisks. Passing an asterisk to `strip()` removes all the asterisks in the string:

```
>>> bad_input = "****Hannah****"
>>> bad_input.strip('*')
'Hannah'
```

If you only want to strip the beginning or end of a string, you can use `rstrip()` or `lstrip()`, respectively. Here, the name Hannah has asterisks before and after it. If we pass an asterisk to `rstrip()`, only asterisks at the end of the string are removed. If we pass an asterisk to `lstrip()`, only asterisks at the beginning of the string are removed.

```
>>> bad_input = "****Hannah****"
>>> bad_input.rstrip('*')
'****Hannah'
>>> bad_input.lstrip('*')
'Hannah****'
```

Searching and Replacing Text

Sometimes, you need to find a piece of text that is located in a string. Strings come with a number of methods that let you search for text. These methods can tell you how many times the text occurs, and let you replace one substring with another.

`count()` returns how many times one string appears in another string. In this example, we're using a rather lengthy bit of text stored in a variable called `long_text`. Let's find how many times the word "the" appears:

```
>>> long_text.count('the')
5
```

Apparently, "the" appears five times.

What if we want to find out where the first instance of "ugly" appears? We can use `find()`. In this example, we want to find where the first instance of the word "ugly" appears in `long_text`.

```
>>> long_text.find('ugly')
25
```

In this example, "ugly" appears starting at the 25th character. A character is one letter, number, space, or symbol.

NOTE

When `find()` Finds Nothing

If `find()` doesn't find anything, it returns -1.

Strings in Python also come with the ability to replace substrings in strings. You can pass two strings to `replace()`, and Python will find all instances of the first string and replace it with the second string.

For example, if we don't like the term "ugly," we can replace it with "meh" by using `replace()` and giving it `'ugly'` and `'meh'` as parameters.

```
>>> long_text.replace('ugly', 'meh')
"Beautiful is better than meh.\n    Explicit is better ...[snip]"
```

NOTE

Zen of Python

Want to see what text I used for this section? In your interpreter, type `import this`. The Zen of Python will print out! This is the main philosophy behind Python, and is one of the Easter eggs in the Python library.

Using Strings in the Real World

In previous hours, we've gone over how Python might help the waiter in our imaginary restaurant. What about the chef? How can strings benefit her?

Most obviously, she can store the specials of the day in a script that can be run later by the waiter. That way, he can run it and see what the specials are without bothering her.

In the following script, the chef has saved a number of specials. She then prints them out in a formatted list of the specials of the day.

```
breakfast_special = "Texas Omelet"
breakfast_notes = "Contains brisket, horseradish cheddar"
lunch_special = "Greek patty melt"
lunch_notes = "Like the regular one, but with tzatziki sauce"
dinner_special = "Buffalo steak"
dinner_notes = "Top loin with hot sauce and blue cheese. NOT BUFFALO MEAT."

print "Today's specials"
print "*"*20
print "Breakfast: ",
print breakfast_special
print breakfast_notes
print
print "Lunch: ",
print lunch_special
print lunch_notes
print
```

```
print "Dinner: ",
print dinner_special
print dinner_notes
```

When the waiter runs it, the following is printed out:

```
Today's specials
********************
Breakfast: Texas Omelet
Contains brisket, horseradish cheddar
Lunch: Greek patty melt
Like the regular one, but with tzatziki sauce
Dinner: Buffalo steak
Top loin with hot sauce and blue cheese. NOT BUFFALO MEAT.
```

If the cook wants to change the specials later, she can edit the first few lines in the file.

Summary

During this hour, you learned that text is stored in something called a string. Python allows you to do certain kinds of math operations on strings, and offers some extra methods for strings, such as removing whitespace.

Q&A

Q. Is there any way to see all of the things I can do with a string without looking it up online?

A. If you want to see everything you can do with strings, type this into your Python shell:

```
>>> s = ""
>>> help(type(s))
```

A list of everything you can do with strings will pop up. Pressing Enter will move you down one line, your up arrow will move you up one line, spacebar will move you down one page, and "q" will close the help menu. Note that this behavior is slightly different in IDLE, where all the text is printed at once.

Incidentally, you can get this screen with any kind of Python data type. If you wanted to find out all the methods that come with the integer type, you could do something like this:

```
>>> s = 1
>>> help(type(s))
```

Q. Why are the methods to remove whitespace from the beginning and end of a string called "right strip" and "left strip"? Why not "beginning" and "end"?

A. In quite a few languages, text isn't printed from left to right. Arabic and Hebrew are both written from right to left, whereas many Eastern scripts are written from top to bottom. "Right" and "left" are more universal than "beginning" and "end".

Q. How big can a string be?

A. That depends on how much memory and hard drive space your computer has. Some languages limit the size of a string, but Python has no hard limit. In theory, one string in your program could fill up your whole hard drive!

Workshop

The Workshop contains quiz questions and exercises to help you solidify your understanding of the material covered. Try to answer all questions before looking at the answers that follow.

Quiz

1. What characters can be stored in strings?
2. What math operators work with strings?
3. What is the backslash character (\) called? What is it used for?

Answers

1. Alphabetic characters, numbers, and symbols can all be stored in strings, as well as whitespace characters such as spaces and tabs.
2. Addition and multiplication operators work with strings.
3. The backslash is called an "escape" and indicates that you want to include some special formatting, such as a tab, new line, a single or double quote, or a backslash.

Exercise

In your program, you're given a string that contains the body of an email. If the email contains the word "emergency," print out "Do you want to make this email urgent?" If it contains the word "joke," print out "Do you want to set this email as non-urgent?"

Index

Symbols

/ (backslash), 43
{} (curly brackets), 55
- (dash), 154
== (double equals), 23-24
= (equals), 23-24
% (percent sign), 58
[] (square brackets), 61
!= (unequal operator), 67
**kwargs, 88, 92
*args, 92

A

absolute value, 21
adding
  colors, to shapes (PyGame), 246
  data, to databases, 202-203
  else, to if statements, 29-30
  items
    to the end of lists, 64
    to repositories, 264-265
  logic, to Flask templates, 235-236
  methods, to classes, 114-115
  strings, together, 40-41
  templates (Flask), 231
    HTML, 231-232
    variables (Flask), 231
    views (Flask), 230
addition, 21
Android applications, creating, 287
appending data to files, 174
applications, 286
apps, Flask, 228-230
arrays, 17
Ascher, David, 292
attending conferences, 288
attributes, OOP (object-oriented programming), 106
avoiding errors, try/except, 32-33

B

backslash (/), 43
Batteries Included, 139
Beazley, David, 292
binary files, 181
blits, 252
blocks
  creating, 28-29
  shells, 29
branches, 267
  creating, 267-269
  merging, 269
breaking out of loops, 74-75
bugs, 273-274
  trying fixes, 279

C

calling functions, 82, 92
choice, 143
circles, drawing (PyGame), 247-248
class inheritance, 130
classes, 133-134
saving classes in files, 130-132
subclasses, 132-133
classes
adding methods to, 114-115
class inheritance, 133-134
comparing values, equality, 126-127
creating basic class statements, 113-114
data types, 125-126
files, 157
greater than, 127-128
instances, 116
__init__() function, 116-118
moving and storing, 118-119
less than, 127-128
OOP (object-oriented programming), 106
overriding default methods, 136
print, 128-130
real world uses, 119-121, 134-136
saving in files, 130-132
cleaning up user input, 54-55
clients, IRC (Internet Relay Chat), 280
code
embedding comments in, 160-162
explaining with docstrings, 162-164
colors, adding, to shapes (PyGame), 246
combining types, 22-23
comma separated values (CSV), 194
command line
converting input(), 51-53
getting information, 49-51
prompts, 51
commands
dir command, 10
mkdir command, 11
pdb debugger, 277
comments, embedding, in code, 160-162
comparing
dictionaries, 98-99
lists, 67
numbers, 23
strings, 42
values, equality, 126-127
comparison operators, 24
conferences, attending, 288
contributing to other projects, 290
contributing to Python, 290
converting input(), command line, 51-53
count(), 63
CSS (cascading style sheets), 291
CSV (comma separated values), 194
curly brackets ({}), 55
cursors, databases, 201
custom dictionaries, creating (JSON), 189-191

D

dash (-), 154
data
adding to databases, 202-203
appending to files, 174
reading from files, 171-172
writing to files, 173-174
data types
classes, 125-126
SQLite, 200
databases, 197
cursors, 201
data, adding, 202-203
deciding when to use, 207-208
deleting, records with DELETE, 216-217
filtering with where, 210
checking for equality, 210-211
checking for inequality, 211
finding non-similar items with NOT LIKE, 212-213
finding similar items with LIKE, 211
querying with greater than and less than, 213
querying, 203-205
real world uses, 205-207
reasons for using, 197-198
sorting, with ORDER BY, 214
SQL (Structured Query Language), 198
real world uses, 217-220
tables, creating, 200-202
unique items, DISTINCT, 215
updating records with UPDATE, 215-216
datetime, 140, 143, 145
time, 144-145
debuggers, pdb, 275-276
default values, setting for functions, 84
DELETE, 216-217
deleting records with DELETE, 216-217
descending ranges, 79
desktop applications, creating (resources), 286-287
dictionaries, 89, 95
comparing, 98-99
creating, 95-97
creating custom, JSON, 189-191
getting information, 97-98
real world uses, 99-101
dictionary, 17
dir command, 10
directories
creating, 177-178
getting information, 175
lists of files, 175-176
moving around, 176-177

DISTINCT, 215
dividing by zero, 23
division, 21
Django, 286
docstrings, explaining code, 162-164
documentation, 159
  docstrings, 162-164
  embedding comments in code, 160-162
  INSTALL, 165
    writing instructions, 166
  README, 164-165
    writing, 166
  real world uses, 167-168
  reasons for good documentation, 159-160
does not equal, 24
double equals (==), 23
double quotes ("), 37
drawing (PyGame)
  circles, 247-248
  text, 252-253
dump(), 186
dynamic content, adding with Jinja, to Flask apps, 234-235

E

elif statements, 30-31
else statements, adding to if statements, 29-30
embedding comments in code, 160-162
__eq__(), 126
equality
  comparing values, classes, 126-127
  filtering with where, 210-211
equals (=), 23-24
errors
  avoiding, try/except, 32-33
  finding,with pdb debugger, 275-276
  locating with traceback, 274-275
escapes, controlling spacing (strings), 43-44
except, 35
  avoiding errors, 32-33
exponents, 21
extend(), 64

F

False, 23
false, variables, 31-32
file directories, including modules from, 152-154
file size, 178-179
file systems, navigating
  Mac, 14-15
  Windows, 10-11
files
  appending data to, 174
  binary files, 181
  creating, 174-175
  getting information, 178
    file size, 178-179
    time accessed, 179
  JSON (JavaScript Object Notation), 185-186
    saving to, 186-187
  opening in write mode, 173
  reading data from, 171-172
  real world uses, 180
  saving classes in, 130-132
  writing data to, 173-174
filtering databases with where, 210-213
finding
  errors with pdb debugger, 275-276
  modules, 145-146
  non-similar items with NOT LIKE, 212-213
  similar items with LIKE, 211
  support
    IRC (Internet Relay Chat), 280-281
    local user groups, 282
    mailing lists, 282
fixes for bugs, trying fixes, 279
Flask, 223-225
  adding views, 230
  creating apps, 228-230
  frameworks, real world uses, 237-238
  installing
    on Macs, 227-228
    in Windows, 225-226
  templates
    adding dynamic content with Jinja, 234-235
    adding logic, 235-236
    creating, 233-234
  templates, adding, 231
    HTML, 231-232
  variables, adding, 231
float, 17
floor division, 21
formatting
  JSON (JavaScript Object Notation), 183-185
  output, 55-56
  strings, 39
    controlling spacing with escapes, 43-44
    removing whitespace, 44-45
    searching and replacing text, 45-46
frameworks, 286
  versus libraries, 223
  real world uses, 237-238
Freenode, 281
functions
  calling, 82, 92
  count(), 63
  creating basic, 81-82
  dump(), 186
  __eq__(), 126
  extend(), 64
  get_receipts(), 193
  getpass() function, 53, 58, 140
  grouping within functions, 88
  has_key(), 97

help(), 163
returning values, 85-86
setting default values, 84
index(), 63
__init__(), 122
input(), 49-53
insert(), 65
is_alpha(), 40
is_digit(), 40
__ne__(), 127
open(), 172
os.getcwd(), 175
os.listdir(), 175-176
os.makedir(), 177
os.makedirs(), 177
os.stat(), 178, 179
os.walk(), 176
passing values to, 82-83
pop(), 96, 101
randint, 141-142
range(), 72
raw_input(), 51
readlines(), 172
real world uses, 89-91
remove(), 65
render(), 252
save_receipts(), 193
scope, 86
creating variables, 86-87
parameters, 87-88
sending parameters, 88-89
__str__(), 128-130
strip(), 45
walk(), 176-177
write(), 173
writelines(), 173

G

game creation competitions, 287
games, PyGame. *See* PyGame
get_receipts(), 193
getpass() function, 53, 58, 140
Git, 261
GitHib, 262
installing, 262
joining GitHib, 261-262
remote repositories, 265-266
repositories
adding items to, 264-265
checking out, 263-264
updating, 266-267
git merge command, 269
GitHib, 262
joining, 261-262
repositories, creating, 263
greater than, 24
classes, 127-128
querying, 213
greater than or equals, 24
grouping, functions, within functions, 88

H

has_key(), 97
Hellmann, Doug, 146, 292
help(), 163
Help Screen, navigating, 163
HTML, 239, 291
templates (Flask), 231-232
HTML tags, 232

I

if statements, 27-28
adding else, 29-30
importing, modules, 154
in, 64
including, modules, from file directories, 152-154
index(), 63
inequality, filtering with where, 211
infinite loops, 76-77
information, storing with variables, 17
__init__() function, 122
classes, instances, 116-118
inline comments, in files, 160
input(), 49-50, 58
converting, command line, 51-53
real world uses, 57
insert(), 65
INSTALL, 165
writing instructions, 166
installations, testing, 15
installing
Flask
on Macs, 227-228
in Windows, 225-226
Git, 262
pip
Macs, 228
Windows, 226
PyGame
Macs, 242-243
Windows, 242
Python
on a Mac, 11
on Windows, 7-8
setuptools
Macs, 227
Windows, 225
SQLite, on Windows, 199-200
text editors
on a Mac, 13-14
on Windows, 9
instances
classes, 116
__init__() function, 116-118
moving and storing, 118-119
OOP (object-oriented programming), 106
integer, 17
Interactive Text Competition, 287
interfaces, 49-50
internet, searching for solutions, 278-279

Internet Relay Chat (IRC), 280-281
Invent Your Own Computer Games with Python, 292
iOS applications, creating, 287
IP addresses, 225
IRC (Internet Relay Chat), 280-281
clients, 280
is_alpha(), 40
is_digit(), 40
items, adding
to the end of lists, 64
to repositories, 264-265
iterating loops, through lists, 73

J

JavaScript, 291
Jinja, adding dynamic content to Flask apps, 234-235
joining GitHib, 261-262
Jones, Brian K., 292
jQuery, 291
json, 140
JSON (JavaScript Object Notation)
custom dictionaries, creating, 189-191
files, 185-186
formatting, 183-185
printing to screen, 187
real world uses, 191-194
saving objects as, 188-189
saving to files, 186-187
Julython, 288

K

keys, 96
Kivy, 287

L

languages, learning, 290-291
Learn Python, 292
Learn Python the Hard Way, 292
less than, 24
classes, 127-128
querying, 213
less than or equals, 24
libraries, versus frameworks, 223
LIKE, finding similar items, 211
LIKE statements, 211
Linux, 16, 288-290
list, 17
list items, skipping to the next list item, loops, 74
lists
adding items to the end of, 64
comparing, 67
creating, 61-63
getting information, 63-64
manipulating, 64-65
math, 65-66
ordering, 66
real world uses, 67-68
lists in lists, 91
lists integrating loops, 73
lists of files, directories, 175-176
local user groups, 282
localhost, 224
locating errors with traceback, 274-275
logic
adding to Flask templates, 235-236
applying to real world problems, 34
long, 17
loops, 71
infinite loops, 76-77
iterating through lists, 73
real world uses, 77-78
repeating, naming loop variables, 73
repeating a set number of times, 71
range of numbers, 72
repeating only when true, 76
infinite loops, 76-77
while loops, 76
skipping to the next list item, 74
variables, 75
while loops, 76

M

Macs
file systems, navigating, 14-15
installing,
Flask, 227-228
PyGame, 242-243
Python, 11
operating systems, determining, 5
running, Python, 12-13
SQLite, 198
text editors, installing, 13-14
mailing lists, 282
main program loops, PyGame, 244-245
managed service providers, 286
manipulating lists, 64-65
Martelli, Alex, 292
math, 20-21
applying to the real world, 23-25
combining types, 22-23
dividing by zero, 23
lists, 65-66
operators, 21
order of operations, 22
Matplotlib, 287
.md, 271
Meetup, 282
merging branches, 269
methods
adding to classes, 114-115
OOP (object-oriented programming), 106
mkdir command, 11

modules, 21, 139
choice, 143
creation of, 147
datetime, 143, 145
time, 144-145
finding, 145-146
importing, 154
including, from file directories, 152-154
random, 140, 142
randint function, 141-142
real world uses, 146-147
uniform, 142-143
moving
around directories, 176-177
instances, 118-119
things around the screen, PyGame, 248-250
to the web, 236
multiplication, 21
strings, 41
music library programs, splitting up, 150-152

N

named parameters, 203
naming
loop variables, 73
variables, 19-20
navigating
file systems
Mac, 14-15
Windows, 10-11
Help Screen, 163
__ne__() function, 127
negation, 21
negative numbers, 68
nicknames, registering, IRC (Internet Relay Chat), 281
NOT LIKE, finding non-similar items, 212-213
Notepad++, 35
numbering, starting at zero, 62
numbers
comparing, 23
length of, 26
storing, in variables, 18-19
NumPy, 287

O

object-oriented programming. *See* OOP (object-oriented programming)
objects
creating objects out of objects, 108-109
defined, 103
planning, 107
real world uses, 110
saving as JSON, 188-189
OOP (object-oriented programming), 103-106, 111
attributes, 106
classes, 106
instances, 106
methods, 106
objects, 104-106
subclasses, 106
vocabulary, 106
open(), 172
opening files, in write mode, 173
operating systems, determining which one you have, 5-7
operators, 21
comparison operators, 24
strings, 42
ORDER BY, sorting, 214
order of operations, math, 22
ordering lists, 66
os, 140
os.getcwd(), 175
os.listdir(), 175-176
os.makedir(), 177
os.makedirs(), 177
os.stat(), 178-179
os.walk(), 176
output
formatting, 55-56
real world uses, 57

P

packages, 139-140
datetime, 140
getpass, 140
json, 140
os, 140
pprint, 140
random, 140
sqlite3, 140
this, 140
packaging, 258
pandas, 287
parameters
named parameters, 203
scope and, 87-88
sending, in functions, 88-89
passing values to functions, 82-83
returning values, 85-86
setting default values, 84
passwords, getting information, 53-54
paths, updating, 225-226
pdb debugger
commands, 277
finding errors with pdb debugger, 275-276
PEP (Python Enhancement Proposal), 26
percent sign (%), 58
pip, 225
installing
Macs, 228
Windows, 226
Planet Python, 292
planning
how to break up programs, 150
objects, 107
Plone, 286
polymorphism. *See* class inheritance

pop(), 96, 101
print, 140
preparations for getting started with Python, 2-3
print, classes, 128-130
print statements, printing, strings, 38
printing
JSON (JavaScript Object Notation), to screen, 187
strings, 38
problems, what to do when you get stuck, 3
programs, splitting
music library programs, 150-152
planning how to break up programs, 150
real world uses, 155-157
reasons for, 149
programs written in Python, 2
prompts, command line, 51
PyGame, 241
drawing, text, 252-253
installing
Macs, 242-243
Windows, 242
moving things around the screen, 248-250
real world uses, 253-257
resources, 253
screens, 243-244
main program loops, 244-245
user input, 245
shapes
adding colors, 246
drawing circles, 247-248
user input, 250-251
Pyglet, 257
PyGUI, 287
Pyjs, 286
Python
installing
on a Mac, 11
on Windows, 7-8
running
on a Mac, 12-13
on Windows, 8-9
Python 2.7, 15
Python 3, 16, 291
Python Anywhere, 286
Python Cookbook, 292
Python Enhancement Proposal (PEP), 26
Python Standard Library by Example, 292
Python Tutor mailing lists, 282
Python.org, 292
PyVideo, 292
PyWeek, 287

Q

querying
databases, 203-205
with greater than and less than, 213
quotes, 37

R

randint function, 141-142
random, 140, 142
choice, 143
randint function, 141-142
uniform, 142-143
range(), 72
range of numbers
descending ranges, 79
loops, 72
Ravenscroft, Anna, 292
raw_input(), 51
reading, data, from files, 171-172
readlines(), 172
README, 164-165
writing, 166
records
deleting with DELETE, 216-217
updating with UPDATE, 215-216
recursion, 93
registering nicknames, IRC (Internet Relay Chat), 281
remote repositories, 265-266
remove(), 65
removing whitespace from strings, 44-45
render(), 252
repeating loops
naming loop variables, 73
range of numbers, 72
set number of times, 71
replacing text, 45-46
repositories
adding items to, 264-265
checking out, 263-264
creating, 263
determining what not to push to the repository, 270-271
remote repositories, 265-266
updating, 266-267
resources
books, 292
Django, 286
Kivy, 287
Plone, 286
PyGUI, 287
Pyjs, 286
Python Anywhere, 286
SciPy, 287
Web2py, 286
websites, 292-293
wxPython, 287
returning values, 85-86
running Python
on a Mac, 12-13
on Windows, 8-9

S

sandboxes, 288
save_receipts(), 193
saving
classes in files, 130-132
JSON (JavaScript Object Notation) to files, 186-187
objects as JSON, 188-189
SciPy, 287
SciPy Library, 287
scope, functions, 86
creating variables, 86-87
parameters, 87-88
screens
printing JSON to, 187
PyGame, 243-244
main program loops, 244-245
user input, 245
searching and replacing text, 45-46
searching internet for solutions, 278-279
sending parameters, functions, 88-89
serve, defined, 224
servers, 224
setuptools, 225
installing
Macs, 227
Windows, 225
shapes, PyGame
adding colors, 246
drawing circles, 247-248
Shaw, Zed, 292
shells, blocks, 29
Shotts, Jr., William E., 290
single quote (', 37
size, testing, functions, 127
skipping to the next list item, loops, 74
sorting, 69
databases with ORDER BY, 214
spaces, 29
spacing, controlling with escapes (strings), 43-44
splitting programs
music library programs, 150-152
planning how to break up programs, 150
real world uses, 155-157
reasons for, 149
SQL (Structured Query Language), 198
real world uses, 217-220
sql statements, 203, 209
SQLAlchemy, 220
SQLite
data types, 200
installing on Windows, 199-200
Macs, 198
testing, 200
users of, 207
sqlite3, 140
square brackets ([]), 61
stack trace, 274
steps, defined, 72
storing
information with variables, 17
instances, 118-119
numbers, in variables, 18-19
__str__() function, 128-130
strings, 17
adding together, 40-41
comparing, 42
creating, 37-38
formatting, 39
controlling spacing with escapes, 43-44
removing whitespace, 44-45
searching and replacing text, 45-46
getting information about, 38-40
multiplication, 41
operators, 42
printing, 38
real world uses, 46-47
strip(), 45
subclasses
class inheritance, 132-133
OOP (object-oriented programming), 106
subtraction, 21
Sweigart, Al, 292
SymPy, 287

T

tables, creating in databases, 200-202
tabs, 29
templates
adding in Flask, 231
HTML, 231-232
creating in Flask, 233-234
Flask
adding dynamic content with Jinja, 234-235
adding logic, 235-236
testing
installations, 15
size, functions for, 127
SQLite, 200
text
drawing, PyGame, 252-253
searching and replacing, 45-46
text editors, installing
on a Mac, 13-14
on Windows, 9
The Hitchhiker's Guide to Python, 293
this, 140
time, 144-145
time accessed, files, 179
timedelta, 145
traceback, locating errors, 274-275
troubleshooting
bugs, 273-274
finding errors with pdb debugger, 275-276
finding support
IRC (Internet Relay Chat), 280-281
local user groups, 282
mailing lists, 282

locating errors with traceback, 274-275
searching the internet for solutions, 278-279
trying fixes, 279
True, 23
true, variables, 31-32
try, avoiding, errors, 32-33
tuples, 17
types of, variables, 17-18

U

Ubuntu, 289-290
unequal operator (!=), 67
uniform, 142-143
unique items, DISTINCT, 215
UPDATE, 215-216
updating
paths, 225-226
records with UPDATE, 215-216
repositories, 266-267
user input
cleaning up, 54-55
PyGame, 245, 250-251

V

values
comparing, equality, 126-127
passing to functions, 82-83
returning values, 85-86
setting default values, 84
returning values, 85-86
variables
adding, in Flask, 231
creating within functions, scope, 86-87
loops, 75
naming, 73
naming, 19-20
storing information, 17
storing numbers, 18-19
true and false, 31-32
types of, 17-18
versioning
branches
creating, 267-269
merging, 269
defined, 259
determining what not to push to the repository, 270-271
Git, 261
GitHib, 262
installing, 262
joining GitHib, 261-262
how it works, 260-261
importance of, 259-260
repositories
adding items to, 264-265
checking out, 263-264
creating, 263
remote repositories, 265-266
updating, 266-267
views, adding in Flask, 230
Virtualbox, 289
virtualenv, 288
virtualenvwrapper, 288

W-X-Y-Z

w+, 175
walk(), 177
Web, moving to, 236
Web Fundamentals, 291
web servers, 224
Web2py, 286
websites, 292-293
creating resources for, 285-286
where, filtering, 210
checking for equality, 210-211
while loops, 76
whitespace, removing, from strings, 44-45
Windows
Flask, installing, 226
installing
PyGame, 242
Python, 7-8
running, Python, 8-9
SQLite, installing, 199-200
text editors, installing, 9
Windows Installer, 7-8
Windows machines, operating systems, determining, 6
write(), 173
write mode, opening files in, 173
writelines(), 173
writing
data to files, 173-174
INSTALL instructions, 166
README, 166
wxPython, 287