

Richard Blum
Christine Bresnahan

Sams **Teach Yourself**

Python Programming for Raspberry Pi

in **24**
Hours

SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



Richard Blum
Christine Bresnahan

Sams **Teach Yourself**

Python

Programming for

Raspberry Pi[®]

in **24**
Hours

SAMS

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

Sams Teach Yourself Python Programming for Raspberry Pi® 24 Hours

Copyright © 2014 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

Raspberry Pi is a trademark of the Raspberry Pi Foundation.

ISBN-13: 978-0-7897-5205-5

ISBN-10: 0-7897-5205-0

Library of Congress Control Number: 2013946052

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(s) 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

Greg Wiegand

Executive Editor

Rick Kughen

Development Editor

Mark Renfrow

Managing Editor

Kristy Hart

Project Editor

Andy Beaster

Copy Editor

Kitty Wilson

Indexer

Tim Wright

Proofreader

Sarah Kearns

Technical Editor

Jason Foster

Publishing Coordinator

Kristen Watterson

Interior Designer

Mark Shirar

Cover Designer

Mark Shirar

Compositor

Nonie Ratcliff

Contents at a Glance

Introduction	1
--------------------	---

Part I: The Raspberry Pi Programming Environment

HOUR 1 Setting Up the Raspberry Pi	7
2 Understanding the Raspbian Linux Distribution	27
3 Setting Up a Programming Environment	45

Part II: Python Fundamentals

HOUR 4 Understanding Python Basics	71
5 Using Arithmetic in Your Programs	97
6 Controlling Your Program	115
7 Learning About Loops	131

Part III: Advanced Python

HOUR 8 Using Lists and Tuples	155
9 Dictionaries and Sets	175
10 Working with Strings	199
11 Using Files	217
12 Creating Functions	239
13 Working with Modules	257
14 Exploring the World of Object-Oriented Programming	277
15 Employing Inheritance	293
16 Regular Expressions	317
17 Exception Handling	337

Part IV: Graphical Programming

HOUR 18 GUI Programming	361
19 Game Programming	387

Part V: Business Programming

HOUR 20 Using the Network	423
21 Using Databases in Your Programming	447
22 Web Programming	469

Part VI: Raspberry Pi Python Projects

HOUR 23 Creating Basic Pi/Python Projects	493
24 Working with Advanced Pi/Python Projects	529

Part VII: Appendix

A Loading the Raspbian Operating System onto an SD Card	553
Index	561

Table of Contents

Introduction	1
Programming with Python	1
Who Should Read This Book?	2
Conventions Used in This Book	3
 Part I: The Raspberry Pi Programming Environment	
 HOUR 1: Setting Up the Raspberry Pi	7
What Is a Raspberry Pi?	7
Acquiring a Raspberry Pi	9
What Raspberry Pi Peripherals Are Necessary?	11
Nice Additional Peripherals	17
Deciding How to Purchase Peripherals	19
Getting Your Raspberry Pi Working	19
Troubleshooting Your Raspberry Pi	24
Summary	25
Q&A	25
Workshop	26
 HOUR 2: Understanding the Raspbian Linux Distribution	27
Learning About Linux	27
Interacting with the Raspbian Command Line	28
Interacting with the Raspbian GUI	33
Summary	43
Q&A	43
Workshop	44
 HOUR 3: Setting Up a Programming Environment	45
Exploring Python	45
Checking Your Python Environment	46
Installing Python and Tools	48

Learning About the Python Interpreter	50
Learning About the Python Interactive Shell	51
Learning About the Python Development Environment Shell	54
Creating Python Scripts	59
Knowing Which Tool to Use and When	65
Summary	66
Q&A	66
Workshop	67

Part II: Python Fundamentals

HOURL 4: Understanding Python Basics	71
Producing Python Script Output	71
Formatting Scripts for Readability	78
Understanding Python Variables	81
Assigning Value to Python Variables	82
Learning About Python Data Types	86
Allowing Python Script Input	88
Summary	94
Q&A	95
Workshop	95
HOURL 5: Using Arithmetic in Your Programs	97
Working with Math Operators	97
Calculating with Fractions	103
Using Complex Number Math	105
Getting Fancy with the <code>math</code> Module	106
Using the <code>NumPy</code> Math Libraries	110
Summary	112
Q&A	113
Workshop	113
HOURL 6: Controlling Your Program	115
Working with the <code>if</code> Statement	115
Grouping Multiple Statements	117

Adding Other Options with the <code>else</code> Statement	118
Adding More Options Using the <code>elif</code> Statement	120
Comparing Values in Python	121
Checking Complex Conditions	126
Negating a Condition Check	127
Summary	128
Q&A	129
Workshop	129
HOUR 7: Learning About Loops	131
Performing Repetitive Tasks	131
Using the <code>for</code> Loop for Iteration	132
Using the <code>while</code> Loop for Iteration	143
Creating Nested Loops	149
Summary	150
Q&A	150
Workshop	151
 Part III: Advanced Python	
HOUR 8: Using Lists and Tuples	155
Introducing Tuples	155
Introducing Lists	160
Using Multidimensional Lists to Store Data	167
Working with Lists and Tuples in Your Scripts	168
Creating Lists by Using List Comprehensions	170
Working with Ranges	170
Summary	172
Q&A	172
Workshop	172
HOUR 9: Dictionaries and Sets	175
Understanding Python Dictionary Terms	175
Exploring Dictionary Basics	176
Programming with Dictionaries	182

Understanding Python Sets	187
Exploring Set Basics	187
Obtaining Information from a Set	189
Modifying a Set	192
Programming with Sets	193
Summary	197
Q&A	197
Workshop	197
HOURL 10: Working with Strings	199
The Basics of Using Strings	199
Using Functions to Manipulate Strings	202
Formatting Strings for Output	209
Summary	215
Q&A	215
Workshop	215
HOURL 11: Using Files	217
Understanding Linux File Structures	217
Opening a File	221
Reading a File	224
Closing a File	230
Writing to a File	231
Summary	237
Q&A	237
Workshop	238
HOURL 12: Creating Functions	239
Utilizing Python Functions in Your Programs	239
Returning a Value	243
Passing Values to Functions	244
Handling Variables in a Function	250
Using Lists with Functions	253
Using Recursion with Functions	254

Summary	255
Q&A	256
Workshop	256
 HOUR 13: Working with Modules	 257
Introducing Module Concepts	257
Exploring Standard Modules	259
Learning About Python Modules	261
Creating Custom Modules	265
Summary	275
Q&A	275
Workshop	276
 HOUR 14: Exploring the World of Object-Oriented Programming	 277
Understanding the Basics of Object-Oriented Programming	277
Defining Class Methods	280
Sharing Your Code with Class Modules	287
Summary	290
Q&A	290
Workshop	290
 HOUR 15: Employing Inheritance	 293
Learning About the Class Problem	293
Understanding Subclasses and Inheritance	294
Using Inheritance in Python	296
Using Inheritance in Python Scripts	302
Summary	314
Q&A	314
Workshop	315
 HOUR 16: Regular Expressions	 317
What Are Regular Expressions?	317
Working with Regular Expressions in Python	319
Defining Basic Patterns	321
Using Advanced Regular Expressions Features	330

Working with Regular Expressions in Your Python Scripts	332
Summary	334
Q&A	335
Workshop	335

HOURL 17: Exception Handling 337

Understanding Exceptions	337
Handling Exceptions	342
Handling Multiple Exceptions	344
Summary	357
Q&A	357
Workshop	357

Part IV: Graphical Programming

HOURL 18: GUI Programming 361

Programming for a GUI Environment	361
Examining Python GUI Packages	363
Using the <code>tkinter</code> Package	364
Exploring the <code>tkinter</code> Widgets	372
Summary	384
Q&A	384
Workshop	384

HOURL 19: Game Programming 387

Understanding Game Programming	387
Learning About Game Frameworks and Libraries	388
Setting Up the <code>PyGame</code> Library	389
Using <code>PyGame</code>	394
Learning More About <code>PyGame</code>	403
Dealing with <code>PyGame</code> Action	408
Summary	418
Q&A	418
Workshop	418

Part V: Business Programming

HOURL 20: Using the Network 423

- Finding the Python Network Modules 423
- Working with Email Servers 424
- Working with Web Servers 432
- Linking Programs Using Socket Programming 438
- Summary 445
- Q&A 445
- Workshop 446

HOURL 21: Using Databases in Your Programming 447

- Working with the MySQL Database 447
- Using the PostgreSQL Database 458
- Summary 466
- Q&A 466
- Workshop 467

HOURL 22: Web Programming 469

- Running a Web Server on the Pi 469
- Programming with the Common Gateway Interface 473
- Expanding Your Python Webpages 475
- Processing Forms 482
- Summary 488
- Q&A 488
- Workshop 488

Part VI: Raspberry Pi Python Projects¹

HOURL 23: Creating Basic Pi/Python Projects 493

- Thinking About Basic Pi/Python Projects 493
- Displaying HD Images via Python 493
- Playing Music 514
- Creating a Special Presentation 521

Summary	526
Q&A	526
Workshop	526
HOUR 24: Working with Advanced Pi/Python Projects	529
Exploring the GPIO Interface	529
Using the <code>RPi.GPIO</code> Module	535
Controlling GPIO Output	536
Detecting GPIO Input	542
Summary	548
Q&A	549
Workshop	549
Part VII: Appendix	
APPENDIX A: Loading the Raspbian Operating System onto an SD Card	553
Windows: Loading Raspbian onto an SD Card	553
Linux: Loading Raspbian onto an SD Card	556
Mac: Loading Raspbian onto an SD Card	558
Index	561

About the Authors

Richard Blum has worked in the IT industry for over 25 years as a network and systems administrator, managing Microsoft, Unix, Linux, and Novell servers for a network with more than 3,500 users. He has developed and teaches programming and Linux courses via the Internet to colleges and universities worldwide. Rich has a master's degree in management information systems from Purdue University and is the author of several Linux books, including *Linux Command Line and Shell Scripting Bible* (coauthored with Christine Bresnahan, 2011, Wiley), *Linux for Dummies*, 9th edition (2009, Wiley), and *Professional Linux Programming* (coauthored with Jon Masters, 2007, Wiley). When he's not busy being a computer nerd, Rich enjoys spending time with his wife, Barbara, and two daughters, Katie Jane and Jessica.

Christine Bresnahan started working in the IT industry more than 25 years ago as a system administrator. Christine is currently an adjunct professor at Ivy Tech Community College in Indianapolis, Indiana, teaching Python programming, Linux system administration, and Linux security classes. Christine produces Unix/Linux educational material and is the author of *Linux Bible*, 8th edition (coauthored with Christopher Negus, 2012, Wiley) and *Linux Command Line and Shell Scripting Bible* (coauthored with Richard Blum, 2011, Wiley). She has been an enthusiastic owner of a Raspberry Pi since 2012.

Dedication

To the Lord God Almighty.

*“I am the vine, you are the branches; he who abides in Me and I in him,
he bears much fruit, for apart from Me you can do nothing.”
—John 15:5*

Acknowledgments

First, all glory, and praise go to God, who through His Son, Jesus Christ, makes all things possible and gives us the gift of eternal life.

Many thanks go to the fantastic team of people at Sams Publishing, for their outstanding work on this project. Thanks to Rick Kughen, the executive editor, for offering us the opportunity to work on this book and keeping things on track. We are grateful to the development editor, Mark Renfrow, who provided diligence in making our work more presentable. Thanks to the production editor, Andy Beaster, for making sure the book was produced. Many thanks to the copy editor, Kitty Wilson, for her endless patience and diligence in making our work readable. Also, we are indebted to our technical editor, Jason Foster, who put in many long hours double-checking all our work and keeping the book technically accurate.

Thanks to Tonya of Tonya Wittig Photography, who created incredible pictures of our Raspberry Pis and was very patient in taking all the photos we wanted for the book. We would also like to thank Carole Jelen at Waterside Productions, Inc., for arranging this opportunity for us and for helping us out in our writing careers.

Christine would also like to thank her student, Paul Bohall, for introducing her to the Raspberry Pi, and her husband, Timothy, for his encouragement to pursue the “geeky stuff” students introduce her to.

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.

This page intentionally left blank

Introduction

Officially launched in February 2012, the Raspberry Pi personal computer took the world by storm, selling out the 10,000 available units immediately. It is an inexpensive credit card-sized exposed circuit board, a fully programmable PC running the free open source Linux operating system. The Raspberry Pi can connect to the Internet, can be plugged into a TV, and costs around \$35.

Originally created to spark schoolchildren's interest in computers, the Raspberry Pi has caught the attention of home hobbyist, entrepreneurs, and educators worldwide. Estimates put the sales figures around 1 million units as of February 2013.

The official programming language of the Raspberry Pi is Python. Python is a flexible programming language that runs on almost any platform. Thus, a program can be created on a Windows PC or Mac and run on the Raspberry Pi and vice versa. Python is an elegant, reliable, powerful, and very popular programming language. Making Python the official programming language of the popular Raspberry Pi was genius.

Programming with Python

The goal of this book is to help guide both students and hobbyists through using the Python programming language on a Raspberry Pi. You don't need to have any programming experience to benefit from this book; we walk through all the necessary steps in getting your Python programs up and running!

Part I, "The Raspberry Pi Programming Environment," walks through the core Raspberry Pi system and how to use the Python environment that's already installed in it. Hour 1, "Setting Up the Raspberry Pi," demonstrates how to set up a Raspberry Pi system, and then in Hour 2, "Understanding the Raspbian Linux Distribution," we take a closer look at Raspbian, the Linux distribution designed specifically for the Raspberry Pi. Hour 3, "Setting Up a Programming Environment," walks through the different ways you can run your Python programs on the Raspberry Pi, and it goes through some tips on how to build your programs.

Part II, "Python Fundamentals," focuses on the Python 3 programming language. Python v3 is the newest version of Python, and is fully supported in the Raspberry Pi. Hours 4 through 7 take you through the basics of Python programming, from simple assignment statements (Hour 4,

“Understanding Python Basics”), arithmetic (Hour 5, “Using Arithmetic in Your Programs”), and structured commands (Hour 6, “Controlling Your Program”), to complex structured commands (Hour 7, “Learning About Loops”).

Hours 8, “Using Lists and Tuples,” and 9, “Dictionaries and Sets,” kick off Part III, “Advanced Python,” showing how to use some of the fancier data structures supported by Python—lists, tuples, dictionaries, and sets. You’ll use these a lot in your Python programs, so it helps to know all about them!

In Hour 10, “Working with Strings,” we take a little extra time to go over how Python handles text strings. String manipulation is a hallmark of the Python programming language, so we want to make sure you’re comfortable with how that all works.

After that primer, we walk through some more complex concepts in Python: using files (Hour 11, “Using Files”), creating your own functions (Hour 12, “Creating Functions”), creating your own modules (Hour 13, “Working with Modules”), object-oriented Python programming (Hour 14, “Exploring the World of Object-Oriented Programming”), inheritance (Hour 15, “Employing Inheritance”), regular expressions (Hour 16, “Regular Expressions”), and working with exceptions (Hour 17, “Exception Handling”).

Part IV, “Graphical Programming,” is devoted to using Python to create real-world applications. Hour 18, “GUI Programming,” discusses GUI programming so you can create your own windows applications, and Hour 19, “Game Programming,” introduces you to the world of Python game programming.

Part V, “Business Programming,” takes a look at some business-oriented applications that you can create. In Hour 20, “Using the Network,” we look at how to incorporate network functions such as email and retrieving data from webpages into your Python programs, Hour 21, “Using Databases in Your Programming,” shows how to interact with popular Linux database servers, and Hour 22, “Web Programming,” demonstrates how to write Python programs that you can access from across the Web.

Part VI, “Raspberry Pi Python Projects,” walks through Python projects that focus specifically on features found on the Raspberry Pi. Hour 23, “Creating Basic Pi/Python Projects,” shows how to use the Raspberry Pi video and sound capabilities to create multimedia projects. Hour 24, “Working with Advanced Pi/Python Projects,” explores connecting your Raspberry Pi with electronic circuits using the General Purpose Input/Output (GPIO) interface.

Who Should Read This Book?

This book is aimed at readers interested in getting the most from their Raspberry Pi system by writing their own Python programs, including these three groups:

- ▶ Students interested in an inexpensive way to learn Python programming.
- ▶ Hobbyists who want to get the most out of their Raspberry Pi system.

- ▶ Entrepreneurs looking for an inexpensive Linux platform to use for application deployment.

If you are reading this book, you are not necessarily new to programming but you may be new to using the Python programming

Conventions Used in This Book

To make your life easier, this book includes various features and conventions that help you get the most out of this book and out of your Raspberry Pi:

Steps	Throughout the book, we've broken many coding tasks into easy-to-follow step-by-step procedures.
Filenames, folder names, and code	These things appear in a <code>monospace</code> font.
Commands	Commands and their syntax use bold .
Menu commands	We use the following style for all application menu commands: <i>Menu</i> , <i>Command</i> , where <i>Menu</i> is the name of the menu you pull down and <i>Command</i> is the name of the command you select. Here's an example: File, Open. This means you select the File menu and then select the Open command.

This book also uses the following boxes to draw your attention to important or interesting information:

BY THE WAY

By the Way boxes present asides that give you more information about the current topic. These tidbits provide extra insights that offer better understanding of the task.

DID YOU KNOW

Did You Know boxes call your attention to suggestions, solutions, or shortcuts that are often hidden, undocumented, or just extra useful.

WATCH OUT!

Watch Out! boxes provide cautions or warnings about actions or mistakes that bring about data loss or other serious consequences.

This page intentionally left blank

This page intentionally left blank

HOUR 4

Understanding Python Basics

What You'll Learn in This Hour:

- ▶ How to produce output from a script
- ▶ Making a script readable
- ▶ How to use variables
- ▶ Assigning value to variables
- ▶ Types of data
- ▶ How to put information into a script

In this hour, you will get a chance to learn some Python basics, such as using the `print` function to display output. You will read about using variables and how to assign them values, and you will gain an understanding of their data types. By the end of the hour, you will know how to get data into a script by using the `input` function, and you will be writing your first Python script!

Producing Python Script Output

Understanding how to produce output from a Python script is a good starting point for those who are new to the Python programming language. You can get instant feedback on your Python statements from the Python interactive interpreter and gently experiment with proper syntax. The `print` function, which you met in Hour 3, “Setting Up a Programming Environment,” is a good place to focus your attention.

Exploring the `print` Function

A *function* is a group of python statements that are put together as a unit to perform a specific task. You can simply enter a single Python statement to perform a task for you.

BY THE WAY

The “New” print Function

In Python v2, `print` is not a function. It became a function when Python v3 was created.

The `print` function’s task is to output items. The “items” to output are correctly called an *argument*. The basic syntax of the `print` function is as follows:

```
print (argument)
```

DID YOU KNOW

Standard Library of Functions

The `print` function is called a *built-in* function because it is part of the Python standard library of functions. You don’t need to do anything special to get this function. It is provided for your use when you install Python.

The *argument* portion of the `print` function can be characters, such as ABC or 123. It can also be values stored in variables. You will learn about variables later in this hour.

Using Characters as print Function Arguments

To display characters (also called *string literals*) using the `print` function, you need to enclose the characters in either a set of single quotes or double quotes. Listing 4.1 shows using a pair of single quotes to enclose characters (a sentence) so it can be used as a `print` function argument.

LISTING 4.1 Using a Pair of Single Quotes to Enclose Characters

```
>>> print ('This is an example of using single quotes.')
This is an example of using single quotes.
>>>
```

Listing 4.2 shows the use of double quotes with the `print` function. You can see that the output that results from both Listing 4.1 and Listing 4.2 does not contain the quotation marks, only the characters.

LISTING 4.2 Using a Pair of Double Quotes to Enclose Characters

```
>>> print ("This is an example of using double quotes.")
This is an example of using double quotes.
>>>
```

BY THE WAY

Choose One Type of Quotes and Stick with It

If you like to use single quotation marks to enclose string literals in a `print` function argument, then consistently use them. If you prefer double quotation marks, then consistently use them. Even though Python doesn't care, it is considered poor form to use single quotes on one `print` function argument and then double quotes on the next. This makes the code hard for humans to read.

Sometimes you need to output a string of characters that contain a single quote to show possession or a contraction. In such a case, you use double quotes around the `print` function argument, as shown in Listing 4.3.

LISTING 4.3 Protecting a Single Quote with Double Quotes

```
>>> print ("This example protects the output's single quote.")
This example protects the output's single quote.
>>>
```

At other times, you need to output a string of characters that contain double quotes, such as for a quotation. Listing 4.4 shows an example of protecting a quote, using single quotes in the argument.

LISTING 4.4 Protecting a Double Quote with Single Quotes

```
>>> print ('I said, "I need to protect my quotation!" and did so.')
I said, "I need to protect my quotation!" and did so.
>>>
```

DID YOU KNOW

Protecting Single Quotes with Single Quotes

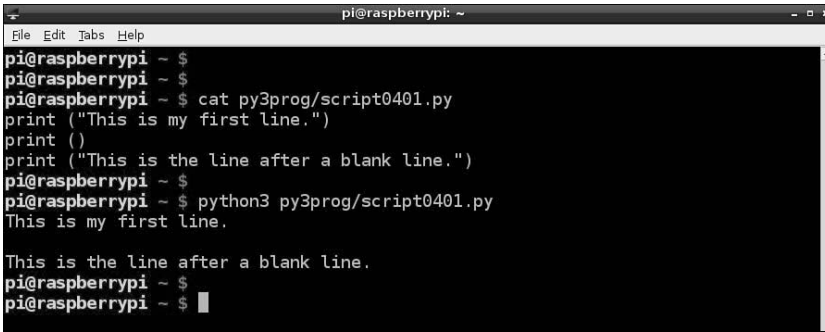
You can also embed single quotes within single quote marks and double quotes within double quote marks. However, when you do, you need to use something called an “escape sequence,” which is covered later in this hour.

Formatting Output with the `print` Function

You can perform various output formatting features by using the `print` function. For example, you can insert a single blank line by using the `print` function with no arguments, like this:

```
print ()
```

The screen in Figure 4.1 shows a short Python script that inserts a blank line between two other lines of output.



```

pi@raspberrypi ~ $
pi@raspberrypi ~ $
pi@raspberrypi ~ $ cat py3prog/script0401.py
print ("This is my first line.")
print ()
print ("This is the line after a blank line.")
pi@raspberrypi ~ $
pi@raspberrypi ~ $ python3 py3prog/script0401.py
This is my first line.

This is the line after a blank line.
pi@raspberrypi ~ $
pi@raspberrypi ~ $ █

```

FIGURE 4.1
Adding a blank line in script output.

Another way to format output using the `print` function is via triple quotes. Triple quotes are simply three sets of double quotes.

Listing 4.5 shows how you can use triple quotes to embed a linefeed character by pressing the Enter key. When the output is displayed, each embedded linefeed character causes the next sentence to appear on the next line. Thus, linefeed moves your output to the next new line. Notice that you cannot see the linefeed character embedded on each line in the code; you can only see its effect in the output.

LISTING 4.5 Using Triple Quotes

```

>>> print ("""This is line one.
... This is line two.
... This is line three.""")
This is line one.
This is line two.
This is line three.
>>>

```

BY THE WAY

But I Prefer Single Quotes

Triple quotes don't have to be three sets of double quotes. You can use three sets of single quotes instead to get the same result!

By using triple quotes, you can also protect single and double quotes that you need to be displayed in the output. Listing 4.6 shows the use of triple quotes to protect both single and double quotes in the same character string.

LISTING 4.6 Using Triple Quotes to Protect Single and Double Quotes

```
>>> print ("Raz said, "I didn't know about triple quotes!" and laughed.")
Raz said, "I didn't know about triple quotes!" and laughed.
>>>
```

Controlling Output with Escape Sequences

An *escape sequence* is a series of characters that allow a Python statement to “escape” from normal behavior. The new behavior can be the addition of special formatting for the output or the protection of characters typically used in syntax. Escape sequences all begin with the backslash (\) character.

An example of using an escape sequence to add special formatting for output is the `\n` escape sequence. The `\n` escape sequence forces any characters listed after it onto the next line of displayed output. This is called a *newline*, and the formatting character it inserts is a linefeed. Listing 4.7 shows an example of using the `\n` escape sequence to insert a linefeed. Notice that it causes the output to be formatted exactly as it was Listing 4.5, with triple quotes.

LISTING 4.7 Using an Escape Sequence to Add a Linefeed

```
>>> print ("This is line one.\nThis is line two.\nAnd this is line three.")
This is line one.
This is line two.
And this is line three.
>>>
```

Typically, the `print` function puts a linefeed only at the end of displayed output. However, the `print` function in Listing 4.7 is forced to “escape” its normal formatting behavior because of the addition of the `\n` escape sequence.

DID YOU KNOW

Quotes and Escape Sequences

Escape sequences work whether you use single quotes, double quotes, or triple quotes to surround your `print` function argument.

You can also use escape sequences to protect various characters used in syntax. Listing 4.8 shows the backslash (\) character used to protect a single quote so that it will not be used in the `print` function's syntax. Instead, the quote is displayed in the output.

LISTING 4.8 Using an Escape Sequence to Protect Quotes

```
>>> print ('Use backslash, so the single quote isn\'t noticed.')
Use backslash, so the single quote isn't noticed.
>>>
```

You can use many different escape sequences in your Python scripts. Table 4.1 shows a few of the available sequences.

TABLE 4.1 A Few Python Escape Sequences

Escape Sequence	Description
\'	Displays a single quote in output.
\"	Displays a double quote in output.
\\	Displays a single backslash in output.
\a	Produces a “bell” sound with output.
\f	Inserts a formfeed into the output.
\n	Inserts a linefeed into the output.
\t	Inserts a horizontal tab into the output.
\u####	Displays the Unicode character denoted by the character's four hexadecimal digits (####).

Notice in Table 4.1 that not only can you insert formatting into your output, you can produce sound as well! Another interesting escape sequence involves displaying Unicode characters in your output.

Now for Something Fun!

Thanks to the Unicode escape sequence, you can print all kinds of characters in your output. You learned a little about Unicode in Hour 3. You can display Unicode characters by using the `\u` escape sequence. Each Unicode character is represented by a hexadecimal number. You can find these hexadecimal numbers at www.unicode.org/charts. There are lots of Unicode characters!

The hexadecimal number for the pi (Π) symbol is 03c0. To display this symbol using the Unicode escape sequence, you must precede the number with `\u` in your `print` function argument. Listing 4.9 displays the pi symbol to output.

LISTING 4.9 Using a Unicode Escape Sequence

```
>>> print ("I love my Raspberry \u03c0!")  
I love my Raspberry  $\pi$ !  
>>>
```

TRY IT YOURSELF ▼

Create Output with the `print` Function

This hour you have been reading about creating and formatting output by using the `print` function. Now it is your turn to try out this versatile Python tool. Follow these steps:


1. If you have not already done so, power up your Raspberry Pi and log in to the system.
2. If you do not have the LXDE GUI started automatically at boot, start it now by typing `startx` and pressing Enter.
3. Open the LXTerminal by double-clicking the LXTerminal icon.
4. At the command-line prompt, type `python3` and press Enter. You are taken to the Python interactive shell, where you can type Python statements and see immediate results.
5. At the Python interactive shell prompt (`>>>`), type `print ('I learned about the print function.')` and press Enter.
6. At the prompt, type `print ('I learned about single quotes.')` and press Enter.
7. At the prompt, type `print ("Double quotes can also be used.')` and press Enter.

BY THE WAY

Multiple Lines with Triple Double Quotes

In steps 8 through 10, you will not be completing the `print` function on one line. Instead, you will be using triple double quotes to allow multiple lines to be entered and displayed.

8. At the prompt, type `print ("\"I learned about things like... and press Enter.`
9. Type `triple quotes,` and press Enter.
10. Type and displaying text on multiple lines.`\"")` and press Enter. Notice that the Python interactive shell did not output the Python print statement's argument until you had fully completed it with the closing parenthesis.
11. At the prompt, type `print ('Single quotes protect "double quotes" in output.')` and press Enter.

- 
12. At the prompt, type `print ("Double quotes protect 'single quotes' in output.")` and press Enter.
 13. At the prompt, type `print ("A backslash protects \"double quotes\" in output.")` and press Enter.
 14. At the prompt, type `print ('A backslash protects \'single quotes\' in output.')` and press Enter. Using the backslash to protect either single or double quotes will allow you to maintain your chosen method of consistently using single (or double quotes) around your print function argument.
 15. At the prompt, type `print ("The backslash character \\ is an escape character.")` and press Enter.
 16. At the prompt, type `print ("Use escape sequences to \n insert a linefeed.")` and press Enter. Notice how part of the sentence, “Use escape sequences to,” is on one line and the end of the sentence “insert a linefeed.” is on another line. This is due to your insertion of the escape sequence `\n` in the middle of the sentence.
 17. At the prompt, type `print ("Use escape sequences to \t\t insert two tabs or")` and press Enter.
 18. At the prompt, type `print ("insert a check mark: \u2714")` and press Enter.

You can do a lot with the print function to display and format output! In fact, you could spend this entire hour just playing with output formatting. However, there are additional important Python basics you need to learn, such as formatting scripts for readability.

Formatting Scripts for Readability

Just as the development environment, IDLE, will help you as your Python scripts get larger, a few minor practices will also be helpful to you. Learn these tips early on, so they become habits as your Python skills grow (and as the length of your scripts grow!).

Long Print Lines

Occasionally you will have to display a very long line of output using the `print` function. It may be a paragraph of instructions you have to provide to your script user. The problem with long output lines is that they make your script code hard to read and the logic behind the script harder to follow. Python is supposed to “fit in your brain.” The habit of breaking up long output lines will help you meet that goal. There are a couple of ways you can accomplish this.

BY THE WAY

A Script User?

You may be one of those people who have never heard the term “user” in association with computers. A *user* is a person who is using the computer or running the script. Sometimes the term “end user” is used instead. You should always keep the “user” in mind when you write your scripts, even if the “user” is just you!

The first way to break up a long output line of characters, is to use something called string concatenation. *String concatenation* takes two or more strings of text and “glues” them together, so they become one string of text. The “glue” in this method is the plus (+) symbol. However, to get this to work properly, you also need to use the backslash (\) to escape out of the normal `print` function behavior of putting a linefeed at the end of a string of characters. Thus, the two items you need are `+\`, as shown in Listing 4.10.

LISTING 4.10 String Concatenation for Long Text Lines

```
>>> print ("This is a really long line of text " +\
... "that I need to display!")
This is a really long line of text that I need to display!
>>>
```

As you can see in Listing 4.10, the two strings are concatenated and displayed as one string in the output. However, there is an even simpler and cleaner method of accomplishing this!

You can forgo the `+\` and simply keep each character string in its own sets of quotation marks. The characters strings will be automatically concatenated by the `print` function! The `print` function handles this perfectly and it is a lot cleaner looking. This method is demonstrated in Listing 4.11.

LISTING 4.11 Combining for Long Text Lines

```
>>> print ("This is a really long line of text "
... "that I need to display!")
This is a really long line of text that I need to display!
>>>
```

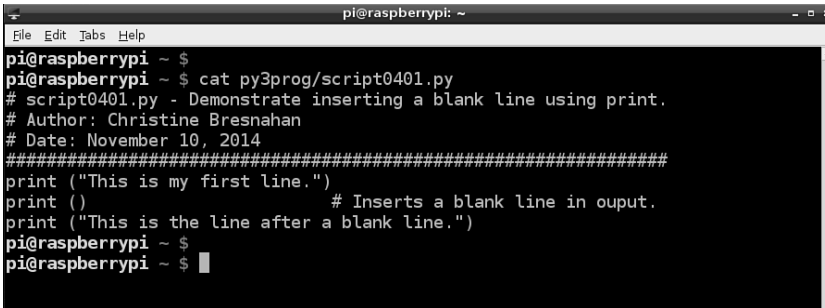
It is always a good rule to keep your Python syntax simple to provide better readability of the scripts. However, sometimes you need to use complex syntax. This is where comments will help you. No, not comments spoken aloud, like “I think this syntax is complicated!” We’re talking about comments that are embedded in your Python script.

Creating Comments

In scripts, *comments* are notes from the Python script author. A comment's purpose is to provide understanding of the script's syntax and logic. The Python interpreter ignores any comments. However, comments are invaluable to humans who need to modify or debug scripts.

To add a comment to a script, you precede it with the pound or hash symbol (#). The Python interpreter ignores anything that follows the hash symbol.

For example, when you write a Python script, it is a good idea to insert comments that include your name, when you wrote the script, and the script's purpose. Figure 4.2 shows an example. Some script writers believe in putting these type of comments at the top of their scripts, while others put them at the bottom. At the very least, if you include a comment with your name as the author in your script, when the script is shared with others, you will get credit for its writing.



```

pi@raspberrypi ~ $
pi@raspberrypi ~ $ cat py3prog/script0401.py
# script0401.py - Demonstrate inserting a blank line using print.
# Author: Christine Bresnahan
# Date: November 10, 2014
#####
print ("This is my first line.")
print ()
print ("This is the line after a blank line.")
# Inserts a blank line in output.
pi@raspberrypi ~ $
pi@raspberrypi ~ $ █

```

FIGURE 4.2
Comments in a Python script.

You can also provide clarity by breaking up sections of your scripts using long lines of the # symbol. Figure 4.2 shows a long line of hash symbols used to separate the comment section from the main body of the script.

Finally, you can put comments at the end of a Python statement. Notice in Figure 4.2 that the `print ()` statement is followed by the comment `# Inserts a blank line in output.` A comment placed at the end of a statement is called an *end comment*, and it provides clarity about that particular line of code.

Those few simple tips will really help you improve the readability of your code. Putting these tips into practice will save you lots of time as you write and modify Python scripts.

Understanding Python Variables

A *variable* is a name that stores a value for later use in a script. A variable is like a coffee cup. A coffee cup typically holds coffee, of course! But a coffee cup can also hold tea, water, milk, rocks, gravel, sand...you get the picture. Think of a variable as a “holder of objects” that you can look at and use in your Python scripts.

BY THE WAY

An Object Reference

Python really doesn't have variables! Instead, they are “object references.” However, for now, just think of them as variables.

When you name your coffee cup...err, variable...you need to be aware that Python variable names are case sensitive. For example, the variables named `CoffeeCup` and `coffeecup` are two different variables. There are other rules associated with creating Python variable names:

- ▶ You cannot use a Python keyword as a variable name.
- ▶ The first character of a variable name cannot be a number.
- ▶ There are no spaces allowed in a variable name.

Python Keywords

The list of Python keywords changes every so often. Therefore, it is a good idea to take a look at the current list of keywords before you start creating variable names. To look at the keywords, you need to use a function that is part of the standard library. However, this function is not built-in, like the `print` function is built-in. You have this function on your Raspbian system, but before you can use it, you need to `import` the function into Python. The function's name is `keyword`. Listing 4.12 shows you how to import into Python and determine keywords.

LISTING 4.12 Determining Python Keywords

```
>>> import keyword
>>> print (keyword.kwlist)
['False', 'None', 'True', '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']
>>>
```

In Listing 4.12, the command `import keyword` brings the `keyword` function into the Python interpreter so it can be used. Then the statement `print (keyword.kwlist)` uses the `keyword` and `print` functions to display the current list of Python keywords. These keywords cannot be used as Python variable names.

Creating Python Variable Names

For the first character in your Python variable name, you must not use a number. The first character in the variable name can be any of the following:

- ▶ A letter a through z
- ▶ A letter A through Z
- ▶ The underscore character (`_`)

After the first character in a variable name, the other characters can be any of the following:

- ▶ The numbers 0 through 9
- ▶ The letters a through z
- ▶ The letters A through Z
- ▶ The underscore character (`_`)

DID YOU KNOW

Using Underscore for Spaces

Because you cannot use spaces in a variable's name, it is a good idea to use underscores in their place, to make your variable names readable. For example, instead of creating a variable name like `coffeecup`, use the variable name `coffee_cup`.

After you determine a name for a variable, you still cannot use it. A variable must have a value assigned to it before it can be used in a Python script.

Assigning Value to Python Variables

Assigning a value to a Python variable is fairly straightforward. You put the variable name first, then an equal sign (`=`), and finish up with the value you are assigning to the variable. This is the syntax:

```
variable = value
```

Listing 4.13 creates the variable `coffee_cup` and assigns a value to it.

LISTING 4.13 Assigning a Value to a Python Variable

```
>>> coffee_cup = 'coffee'
>>> print (coffee_cup)
coffee
>>>
```

As you see in Listing 4.13, the `print` function can output the value of the variable without any quotation marks around it. You can take output a step further by putting a string and a variable together as two `print` function arguments. The `print` function knows they are two separate arguments because they are separated by a comma (`,`), as shown in Listing 4.14.

LISTING 4.14 Displaying Text and a Variable

```
>>> print ("My coffee cup is full of", coffee_cup)
My coffee cup is full of coffee
>>>
```

Formatting Variable and String Output

Using variables adds additional formatting issues. For example, the `print` function automatically inserts a space whenever it encounters a comma in a statement. This is why you do not need to add a space at the end of `My coffee cup is full of`, as shown in Listing 4.14. There may be times, however, when you want something else besides a space to separate a string of characters from a variable in the output. In such a case, you can use a separator in your statement. Listing 4.15 uses the `sep` separator to place an asterisk (`*`) in the output instead of a space.

LISTING 4.15 Using Separators in Output

```
>>> coffee_cup = 'coffee'
>>> print ("I love my", coffee_cup, "!", sep='*')
I love my*coffee*!
>>>
```

Notice you can also put variables in between various strings in your `print` statements. In Listing 4.15, four arguments are given to the `print` function:

- ▶ The string `"I love my"`
- ▶ The variable `coffee_cup`

- ▶ The string "!"
- ▶ The separator designation '*'

The variable `coffee_cup` is between two strings. Thus, you get two asterisks (*), one between each argument to the `print` function. Mixing strings and variables in the `print` function gives you a lot of flexibility in your script's output.

Avoiding Unassigned Variables

You cannot use a variable until you have assigned a value to it. A variable is created when it is assigned a value and not before. Listing 4.16 shows an example of this.

LISTING 4.16 Behavior of an Unassigned Variable

```
>>> print (glass)
Traceback (most recent call last):
File "<stdin>", line 1, in <module> Name
Error: name 'glass' is not defined
>>>
>>> glass = 'water'
>>> print (glass)
water
>>>
```

Assigning Long String Values to Variables

If you need to assign a long string value to a variable, you can break it up onto multiple lines by using a couple methods. Earlier in the hour, in the “Formatting Scripts for Readability” section, you looked at using the `print` function with multiple lines of outputted text. The concept here is very similar.

The first method involves using string concatenation (+) to put the strings together and an escape character (\) to keep a linefeed from being inserted. You can see in Listing 4.17 that two long lines of text were concatenated together in the assignment of the variable `long_string`.

LISTING 4.17 Concatenating Text in Variable Assignment

```
>>> long_string = "This is a really long line of text" + \
... " that I need to display!"
>>> print (long_string)
This is a really long line of text that I need to display!
>>>
```

Another method is to use parentheses to enclose your variable's value. Listing 4.18 eliminates the +\ and uses parentheses on either side of the entire long string in order to make it into one long string of characters.

LISTING 4.18 Combining Text in Variable Assignment

```
>>> long_string = ("This is a really long line of text"
... " that I need to display!")
>>> print (long_string)
This is a really long line of text that I need to display!
>>>
```

The method used in Listing 4.18 is a much cleaner method. It also helps improve the readability of the script.

BY THE WAY

Assigning Short Strings to Variables

You can use parentheses for assigning short strings to variables, too! This is especially useful if it helps you improve the readability of your Python script.

More Variable Assignments

The value of a variable does not have to only be a string of characters; it can also be a number. In Listing 4.19, the number of cups consumed of a particular beverage are assigned to the variable `cups_consumed`.

LISTING 4.19 Assigning a Numeric Value to a Variable

```
>>> coffee_cup = 'coffee'
>>> cups_consumed = 3
>>> print ("I had", cups_consumed, "cups of",
... coffee_cup, "today!")
I had 3 cups of coffee today!
>>>
```

You can also assign the result of an expression to a variable. The equation $3+1$ is completed in Listing 4.20, and then the value 4 is assigned to the variable `cups_consumed`.

LISTING 4.20 Assigning an Expression Result to a Variable

```
>>> coffee_cup = 'coffee'
>>> cups_consumed = 3 + 1
>>> print ("I had", cups_consumed, "cups of",
... coffee_cup, "today!")
I had 4 cups of coffee today!
>>>
```

You will learn more about performing mathematical operations in Python scripts in Hour 5, “Using Arithmetic in Your Programs.”

Reassigning Values to a Variable

After you assign a value to a variable, the variable is not stuck with that value. It can be reassigned. Variables are called *variables* because their values can be varied. (Say that three times fast.)

In Listing 4.21, the variable `coffee_cup` has its value changed from `coffee` to `tea`. To reassign a value, you simply enter the assignment syntax with a new value at the end of it.

LISTING 4.21 Reassigning a Variable

```
>>> coffee_cup = 'coffee'
>>> print ("My cup is full of", coffee_cup)
My cup is full of coffee
>>> coffee_cup = 'tea'
>>> print ("My cup is full of", coffee_cup)
My cup is full of tea
>>>
```

DID YOU KNOW

Variable Name Case

Python script writers tend to use all lowercase letters in the names of variable whose values might change, such as `coffee_cup`. For variable names that are never reassigned values, all uppercase letters are used (for example, `PI = 3.14159`). The unchanging variables are called *symbolic constants*.

Learning About Python Data Types

When a variable is created by an assignment such as `variable = value`, Python determines and assigns a data type to the variable. A *data type* defines how the variable is stored and the

rules governing how the data can be manipulated. Python uses the value assigned to the variable to determine its type.

So far, this hour has focused on strings of characters. When the Python statement `coffee_cup = 'tea'` was entered, Python saw the characters in quotation marks and determined the variable `coffee_cup` to be a *string literal* data type, or `str`. Table 4.2 lists a few of the basic data types Python assigns to variables.

TABLE 4.2 Python Basic Data Types

Data Type	Description
<code>float</code>	Floating-point number
<code>int</code>	Integer
<code>long</code>	Long integer
<code>str</code>	Character string or string literal

You can determine what data type Python has assigned to a variable by using the `type` function. In Listing 4.22, you can see that the variables have been assigned two different data types.

LISTING 4.22 Assigned Data Types for Variables

```
>>> coffee_cup = 'coffee'
>>> type (coffee_cup)
<class 'str'>
>>> cups_consumed = 3
>>> type (cups_consumed)
<class 'int'>
>>>
```

Python assigned the data type `str` to the variable `coffee_cup` because it saw a string of characters between quotation marks. However, for the `cups_consumed` variable, Python saw a whole number, and thus it assigned it the integer data type, `int`.

DID YOU KNOW

The `print` Function and Data Types

The `print` function assigns to its arguments the string literal data type, `str`. It does this for anything that is given as an argument, such as quoted characters, numbers, variables values, and so on. Thus, you can mix data types in your `print` function argument. The `print` function will just convert everything to a string literal data type and spit it out to the display.

Making a small change in the `cups_consumed` variable assignment statement causes Python to change its data type. In Listing 4.23, the number assigned to `cups_consumed` is reassigned from 3 to 3.5. This causes Python to reassign the data type to `cups_consumed` from `int` to `float`.

LISTING 4.23 Changed Data Types for Variables

```
>>> cups_consumed = 3
>>> type (cups_consumed)
<class 'int'>
>>> cups_consumed = 3.5
>>> type (cups_consumed)
<class 'float'>
>>>
```

You can see that Python does a lot of the “dirty work” for you. This is one of the many reasons Python is so popular.

Allowing Python Script Input

There will be times that you need a script user to provide data into your script from the keyboard. In order to accomplish this task, Python provides the `input` function. The `input` function is a built-in function and has the following syntax:

```
variable = input (user prompt)
```

In Listing 4.24, the variable `cups_consumed` is assigned the value returned by the `input` function. The script user is prompted to provide this information. The prompt provided to the user is designated in the `input` function as an argument. The script user inputs an answer and presses the Enter key. This action causes the `input` function to assign the answer 3 as a value to the variable `cups_consumed`.

LISTING 4.24 Variable Assignment via Script Input

```
>>> cups_consumed = input("How many cups did you drink? ")
How many cups did you drink? 3
>>> print ("You drank", cups_consumed, "cups!")
You drank 3 cups!
>>>
```

For the user prompt, you can enclose the prompt’s string characters in either single or double quotes. The prompt is shown enclosed in double quotes in Listing 4.24’s `input` function.

BY THE WAY

Be Nice to Your Script User

Be nice to the user of your script, even if it is just yourself. It is no fun typing in an answer that is “squished” up against the prompt. Add a space at the end of each prompt to give the end user a little breathing room for prompt answers. Notice in the input function in Listing 4.24 that there is a space added between the question mark (?) and the enclosing double quotes.

The `input` function treats all input as strings. This is different from how Python handles other variable assignments. Remember that if `cups_consumed = 3` were in your Python script, it would be assigned the data type integer, `int`. When using the `input` function, as shown in Listing 4.25, the data type is set to string, `str`.

LISTING 4.25 Data Type Assignments via Input

```
>>> cups_consumed = 3
>>> type (cups_consumed)
<class 'int'>
>>> cups_consumed = input("How many cups did you drink? ")
How many cups did you drink? 3
>>> type (cups_consumed)
<class 'str'>
>>>
```

To convert variables which are input from the keyboard, from strings, you can use the `int` function. The `int` function will convert a number from a string data type to an integer data type. You can use the `float` function to convert a number from a string to a floating-point data type. Listing 4.26 shows how to convert the variable `cups_consumed` to an integer data type.

LISTING 4.26 Data Type Conversion via the int Function

```
>>> cups_consumed = input ("How many cups did you drink? ")
How many cups did you drink? 3
>>> type (cups_consumed)
<class 'str'>
>>> cups_consumed = int(cups_consumed)
>>> type (cups_consumed)
<class 'int'>
>>>
```

You can get really tricky here and use a nested function. *Nested functions* are functions within functions. The general format, is as follows:

```
variable = functionA(functionB(user_prompt))
```

Listing 4.27 uses this method to properly change the input data type from a string to an integer.

LISTING 4.27 Using Nested Functions with `input`

```
>>> cups_consumed = int(input("How many cups did you drink? "))
How many cups did you drink? 3
>>> type (cups_consumed)
<class 'int'>
>>>
```

Using nested functions makes a Python script more concise. However, the trade-off is that the script is a little harder to read.

▼ TRY IT YOURSELF

Explore Python Input and Output with Variables

You are now going to explore Python input and output using variables. In the following steps, you will write a script to play with, instead of using the interactive Python shell:

1. If you have not already done so, power up your Raspberry Pi and log in to the system.
2. If you do not have the LXDE GUI started automatically at boot, start it now by typing `startx` and pressing Enter.
3. Open the LXTerminal by double-clicking the LXTerminal icon.
4. At the command-line prompt, type `nano py3prog/script0402.py` and press Enter. The command puts you into the nano text editor and creates the file `py3prog/script0402.py`.
5. Type the following code into the nano editor window, pressing Enter at the end of each line:

```
# script0402.py - My first real Python script.
# Written by <your name here>
# Date: <today's date>
#
##### Define Variables #####
#
amount = 4           #Number of vessels.
vessels = 'glasses' #Type of vessels used.
liquid = 'water'     #What is contained in the vessels.
location = 'on the table' #Location of vessels.
#
##### Output Variable Description #####
#
```

```
print ()
#
print ("The variables are as follows:")
#
print ("name: amount", "data type:", type (amount), "value:", amount)
#
print ("name: vessels", "data type:", type (vessels), "value:", vessels)
#
print ("name: liquid", "data type:", type (liquid), "value:", liquid)
#
print ("name: location", "data type:", type (location), "value:", location)
print ()
#
##### Output Sentence Using Variables #####
#
print ("There are", amount, vessels, "full of", liquid, location, end='\n')
print ()
```

BY THE WAY

Be Careful!

Be sure to take your time here and avoid making typographical errors. Double-check and make sure you have entered the code into the nano text editor window as shown above. You can make corrections by using the Delete key and the up- and down-arrow keys.

6. Write out the information you just typed in the text editor to the script by pressing Ctrl+O. The script file name will show along with the prompt `File name to write`. Press Enter to write out the contents to the `script0402.py` script.
7. Exit the nano text editor by pressing Ctrl+X.
8. Type `python3 py3prog/script0402.py` and press Enter to run the script. If you encounter any errors, note them so you can fix them in the next step. You should see output like the output shown in Figure 4.3. The output is okay, but it's a little sloppy. You can clean it up in the next step.

```

pi@raspberrypi ~ $
pi@raspberrypi ~ $
pi@raspberrypi ~ $ python3 py3prog/script0402.py
This script has four variables pre-defined in it.

The variables are as follows:
name: amount data type: <class 'int'> value: 4
name: vessels data type: <class 'str'> value: glasses
name: liquid data type: <class 'str'> value: water
name: location data type: <class 'str'> value: on the table

There are 4 glasses full of water on the table.

pi@raspberrypi ~ $

```

FIGURE 4.3

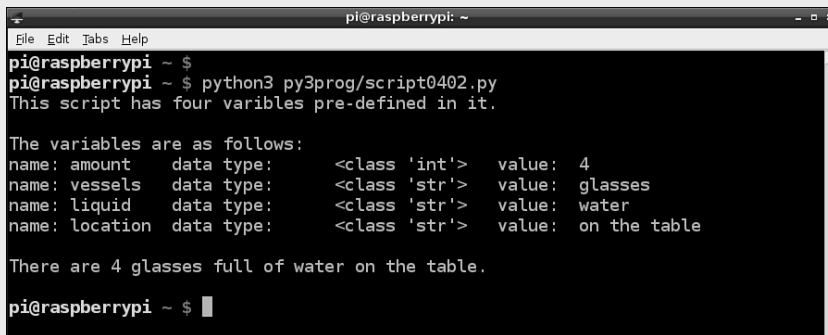
Output for the Python script `script0402.py`.

9. At the command-line prompt, type `nano py3prog/script0402.py` and press Enter. The command puts you into the nano text editor, where you can modify the `script0402.py` script.
10. Go to the Output Variable Description portion of the script and add a separator to the end of each line. The lines of code to be changed and how they should look when you are done are shown here:


```

print ("name: amount", "data type:", type (amount), "value:", amount, sep='\t')
#
print ("name: vessels", "data type:", type (vessels), "value:", vessels,
      ↪sep='\t')
#
print ("name: liquid", "data type:", type (liquid), "value:", liquid, sep='\t')
#
print ("name: location", "data type:", type (location), "value:",
      ↪location,sep='\t')

```
11. Write out the modified script by pressing Ctrl+O. Press Enter to write out the contents to the `script0402.py` script.
12. Exit the nano text editor by pressing Ctrl+X.
13. Type `python3 py3prog/script0402.py` and press Enter to run the script. You should see output like the output shown in Figure 4.4. Much neater!



```

pi@raspberrypi ~ $
pi@raspberrypi ~ $ python3 py3prog/script0402.py
This script has four variables pre-defined in it.

The variables are as follows:
name: amount      data type:      <class 'int'>   value: 4
name: vessels     data type:      <class 'str'>  value: glasses
name: liquid      data type:      <class 'str'>  value: water
name: location    data type:      <class 'str'>  value: on the table

There are 4 glasses full of water on the table.

pi@raspberrypi ~ $ █

```

FIGURE 4.4

The script0402.py output, properly tabbed.

14. To try adding some input into your script, at the command-line prompt, type `nano py3prog/script0402.py` and press Enter.
15. Go to the bottom of the script and add the additional Python statements shown here:

```

#
##### Get Input #####
#
print ()
print ("Now you may change the variables' values.")
print ()
#
amount=int(input("How many vessels are there? "))
print ()
#
vessels = input("What type of vessels are being used? ")
print ()
#
liquid = input("What type of liquid is in the vessel? ")
print ()
#
location=input("Where are the vessels located? ")
print ()
#
##### Display New Input to Output #####
#
print ("So you believe there are", amount, vessels, "of", liquid, location,
➤end='\n')
print ()
#
##### End of Script #####

```

16. Write out the modified script by pressing Ctrl+O. Press Enter to write out the contents to the `script0402.py` script.
17. Exit the nano text editor by pressing Ctrl+X.
18. Type `python3 py3prog/script0402.py` and press Enter to run the script. Answer the prompts any way you want. (You are supposed to be having fun here!) Figure 4.5 shows what your output should look like.

```

pi@raspberrypi ~ $ python3 py3prog/script0402.py
This script has four variables pre-defined in it.

The variables are as follows:
name: amount      data type:    <class 'int'>   value: 4
name: vessels     data type:    <class 'str'>  value: glasses
name: liquid      data type:    <class 'str'>  value: water
name: location    data type:    <class 'str'>  value: on the table

There are 4 glasses full of water on the table.

Now you may change the variables' values.
How many vessels are there? 99
What type of vessels are being used? bottles
What type of liquid is in the vessel? tea
Where are the vessels located? on the wall
So you believe there are 99 bottles of tea on the wall.

pi@raspberrypi ~ $

```

FIGURE 4.5
The complete `script0402.py` output.

Run this script as many times as you want. Experiment with the various types of answers you put in and see what the results are. Also try making some minor modifications to the script and see what happens. Experimenting and playing with your Python script will enhance your learning.

Summary

In this hour, you got a wonderful overview of Python basics. You learned about output and formatting output from Python; creating legal variable names and assigning values to variables; and various data types and when they are assigned by Python. You explored how Python can handle input from the keyboard and how to convert the data types of the variables receiving

that input. Finally, you got to play with your first Python script. In Hour 5, your Python exploration will continue as you delve into mathematical algorithms with Python.

Q&A

Q. Can I do any other kind of output formatting besides what I learned about in this chapter?

A. Yes, you can also use the `format` function, which is covered in Hour 5.

Q. Which is better to use with a `print` function, double quotes or single quotes?

A. Neither one is better than the other. Which one you use is a personal preference. However, whichever one you choose, it's best to consistently stick with it.

Q. Bottles of tea on the wall?!

A. This is a family-friendly book. Feel free to modify your answers to `script0402.py` to your liking.

Workshop

Quiz

1. The `print` function is part of the Python standard library and is considered a built-in function. True or false?
2. When is a variable created and assigned a data type?
3. Which of the following is a valid Python data type?
 - a. `int`
 - b. `input`
 - c. `print`

Answers

1. True. The `print` function is a built-in function of the standard library. There is no need to import it.
2. A variable is created and assigned a data type when it is assigned a value. The value and data type for a variable can be changed with a reassignment.
3. `int` is a Python data type. `input` and `print` are both built-in Python *functions*.

This page intentionally left blank

Index

Symbols

- // (floor division operator), 98
- { } placeholder, 102-103
- * (asterisk), as pattern for regular expressions, 329
- " (double quotes), print function syntax, 72-73
- # (hash symbol), 80
- | (pipe symbol), 331
- + (plus sign), 330
- ? (question mark), 330
- ' (single quotes), print function syntax, 72-73

A

- absolute directory references, 219
- accessing
 - data from dictionaries, 178-180
 - with get operation, 178-179
 - with for loops, 179
 - with sorted() function, 179-180
 - data in tuples, 157
 - GUI, 29

- information from sets, 189-192
 - set difference, 190-191
 - set intersection, 190
 - set membership, 189
 - set union, 189-190
 - symmetric set
 - difference, 191
- range of values in tuples, 157-158
- accessor methods, 281-283**
- acquiring Raspberry Pi, 9-11**
- .add operation, 188-189**
- adding**
 - comments to Python scripts, 80
 - elements to sets, 188-189
 - line feeds with escape sequences, 75-76
 - new data values to lists, 164-165
 - subclasses to object module file, 298-300
 - widgets to windows, 366-370
- advanced regular expression features, 330-332**
- Akerman, Dave, 9**
- allowing input in Python scripts, 88-89**
- altering strings with functions, 202-204**
 - string-joining functions, 205
 - string-splitting functions, 204-205
 - string-testing functions, 205-206

- analog televisions, connecting Raspberry Pi to, 15**
- anchor characters as pattern for regular expressions, 324-325**
- AND operators, 98**
- Apache web server**
 - CGI, 473-475
 - installing, 470-471
- append() function, 164**
- arguments**
 - passing, 244-246
 - print function, 72-73
- array() function, 111**
- arrays**
 - associative arrays, 175-176
 - creating with NumPy module, 111-112
- ASCII code, converting to string value, 200**
- assigning**
 - calculation results to variables, 101
 - value to variables, 82-86
 - long string values, 84-85
 - number values, 85-86
- associative arrays, 175-176**
- asterisk (*), as pattern for regular expressions, 329**
- asynchronous events, 547-548**
- audio, adding to games, 407**
- available modules, exploring, 264-265**

B

- backslash, 75**
- base classes, 294**
- binary operators, 98**
- bind() function, 441**
- Blender3D, 389**
- blinking the LED, 540-541**
- Boolean comparisons, 124-125**
- booting**
 - directly to GUI, 35-36
 - Raspberry Pi, 21-23
 - troubleshooting, 24-25
- break statement, terminating infinite loops, 146-147**
- Broadcom chip, 529**
- building**
 - Gertboard circuit, 538-539
 - Pi Cobbler circuit, 537-538
- built-in modules, 258**
- Button widget, 373**
- buying Raspberry Pi, 9-11**
 - required peripherals, 11-16
 - keyboard, 16
 - output display, 15-16
 - power supply, 13-15
 - SD card, 12-13

C

calculations

- complex number math, 105-106
- floating-point accuracy, 102
- Fraction object, 103-105
- variables, 100-101

calendar command, 31

camel Case, 194

cases, selecting, 17-18

cat -n command, 338-339

centering photos, 503-507

CGI (Common Gateway Interface), 473-475

- debugging, 480-482

cgi module, 484-487

changing

- order of operations, 100
- passwords, 32-33

character classes

- negating, 327-328
- as pattern for regular expressions, 326-327

charitable organizations

- Raspberry Pi Foundation, 9

Checkbutton widget, 373-375

classes, 278-279

- the class problem, 293-294
- default attribute values, 279-280
- deleting, 285
- documenting, 286
- inheritance, 295-302
- instantiating, 279

methods, 280-287

- accessor methods, 281-283
- constructors, 283-284
- mutator methods, 280-281

modules, creating, 287-289

output, customizing, 284-285

subclasses, 294-295, 308-314

- creating, 297-298
- polymorphism, 306
- storing in its own object module file, 301-302

superclasses, 294

client programs

- creating, 441-444
- running, 443-445

client/server programs, 438-439

closing files, 228-231

cocos2D, 389

color coding syntax in development environment, 53

combining condition checks, 126-127

command line, 29-33

- entering commands, 31-33
- whoami command, 29

commands

- calendar, 30
- cat -n, 338-339
- for creating Python scripts, 65

entering at command line, 31-33

ls, 31

mkdir, 31

nano, 65

pwd, 31

reboot, 31

reviewing, 54

for running Python scripts, 65

structured commands, 115

- if statement, 115-117

sudo, 32

for testing Python statements, 65

whoami, 29

comments, adding to scripts, 80

comparing

- discard and remove operations, 193

Model A and Model B, 10-11

Python v2 and Python v3, 46

tuples, 124

values

- Boolean comparisons, 124-125

- numeric comparisons, 122

- string comparisons, 122-124

compiled programming languages, 50

complex numbers, 105-106

- creating, 105

concatenating

- lists, 165-166
- tuples, 160

condition checks

- combining, 126-127
- negating, 127-128

configuring

- direct bootup to GUI, 35-36
- U.S. keyboard, 49-50

connecting

- to GPIO, 530-534
 - via Gertboard, 533-534
 - via Pi Cobbler, 532-533
- to MySQL databases, 452
- to PostgreSQL databases, 463-464
- Raspberry Pi to analog televisions, 15
- Raspberry Pi to modern output displays, 16

constructors, 283-284**controlling**

- music playback, 517-521
- output with escape sequences, 75-76

converting

- ASCII code to string value, 200
- variable data types, 89-90

count() function, 166**creating**

- arrays with NumPy module, 111-112
- class modules, 287-289

client programs, 441-444

complex numbers, 105

custom modules, 264-275

dictionaries, 176

factorial functions, 254-255

fractions, 103-105

functions, 240

image presentation script, 495-496

lists, 160-161

multidimensional lists, 167

MySQL databases, 449-450

output with print function, 78

PostgreSQL databases, 459-460

Python scripts

- commands used for, 65
- in IDLE, 62-63
- with nano, 64-65

server programs, 439-441

sets, 187-188

shapes, 404-407

strings, 200-201

subclasses, 297-298

tuples, 155-156

web forms, 483-484

windows, 364-366

CSSSelector() method, 435-436**custom modules**

- creating, 264-272, 272-273
- testing, 267-268, 272

customizing class output, 284-285**D****dash shell, 29****data types, 86-88**

- converting, 89-90
- in NumPy module, 110-111
- range, 170-171
- tuples, 155-160
 - accessing data in, 157
 - comparison operations, 158-159
 - concatenating, 160
 - creating, 155-156
 - maximum value, identifying, 159-160
 - minimum value, identifying, 159-160
 - number of values in, identifying, 159
 - range of values, accessing, 157-158

databases, 447

- MySQL, 447-458
 - connecting to, 452
 - creating, 449-450

- environment, setting
 - up, 448-452
- inserting data, 455-457
- installing, 448
- querying data, 457-458
- tables, creating,
 - 451-452
- user accounts, creating,
 - 450-451
- PostgreSQL, 447, 458-466
 - connecting to, 463-464
 - creating, 459-460
 - environment, setting
 - up, 459-463
 - inserting data, 464-465
 - querying data, 465-466
 - tables, creating,
 - 461-463
 - user accounts, creating,
 - 460-461
 - publishing data on the
 - web, 478
 - relational databases, 451
- Debian, 28**
- debugger features (IDLE), 56**
- debugging CGI, 480-482**
- deep copy, 184**
- default class attribute values, 279-280**
- default parameter values, setting, 246-247**
- defining**
 - classes, 278-279
 - event handlers, 370-372
 - functions, 240-243
- deleting**
 - class instances, 285
 - list values, 163
- designating open() function mode, 222-223**
- desktop environments (Linux), changing, 34**
- detecting GPIO input, 542**
- development environment shell (Python), 50, 53-57. See also IDLE**
- dictionaries, 175-176**
 - accessing data from,
 - 178-180
 - with get operation,
 - 178-179
 - with for loops, 179
 - with sorted() function,
 - 179-180
 - creating, 176
 - deep copy, 184
 - managing, 181-182
 - populating, 176-178
 - programming with,
 - 182-187
 - shallow copy, 184
 - updating, 180-181
 - values, retrieving, 249-250
- Digital Clock icon (LXPanel), 39**
- dir() function, 262**
- directories**
 - absolute directory
 - references, 219
 - home directory, 31
 - Linux directory structure,
 - 217-219
 - managing, 220-221
 - path directories, checking,
 - 270-271
 - present working
 - directory, 31
 - production directory,
 - moving modules to,
 - 268-270
 - relative directory
 - references, 219
 - test directories, creating
 - modules in, 266-267
 - upside-down trees, 217
- discard operation, 192**
- displaying**
 - numbers, 102-103
 - text on game screen,
 - 399-403
 - Unicode, 76-77
- distributions (Linux), 27**
- documentation for Debian distribution, 28**
- documenting classes, 286**
- dot character as pattern for regular expressions, 325-326**
- double quotes ("), print function syntax, 72-73**
- downloading Raspbian, 20**
- dpi (dots per inch), 494**
- drawing shapes, 404-407**
- duty cycle, 541**
- dynamic webpages, 476-480**

E**ehlo() method, 429****elements**

- adding to sets, 188-189
- set difference, 190-191
- set intersection, 190
- set membership, 189
- set union, 189-190
- symmetric set
 - difference, 191

elif statement, 120-121**else clause (while loops), 145****else statement, 118-120****email servers, 423-432**

- Linux, modular email
 - environment, 425-427
- remote email servers, 428
- RFC 2822 email standard, 429
- sending messages, 430-432
- smtpplib module, 427-429

entering commands at command line, 31-33**Entry widget, 397-398****equations**

- complex number math, 105-106
- floating-point accuracy in calculations, 102
- Fraction object, 103-105
- numbers, displaying, 102-103
- variables, 100-101

errors

- generic exception handling, 350
- runtime error exceptions, 340-342
- syntactical error
 - exceptions, 337-339

escape sequences

- line feeds, adding, 75-76
- output, controlling, 75-76
- quotes, protecting, 76
- Unicode, displaying, 76-77

etree methods, parsing HTML elements, 434-435**evaluating function results, 125-126****event handling, 370-372, 403-404****event-driven programming, 362-363****exception handling, 337, 342-344. See also troubleshooting**

- generic exceptions, 350
- for multiple exceptions, 344-352
- runtime error exceptions, 340-342
- syntactical error
 - exceptions, 337-339
- try except statement, 344
 - options, 350-352

exit () statement, 57**exiting**

- IDLE, 57
- Python interactive shell, 52

exploring

- available modules, 264-265
- inheritance, 308-314
- Python interactive shell, 54
- subclasses, 308-314

expressions, assigning results of to variables, 85-86**extend() function, 166****extracting data from lists, 161-162****extracting data from webpages, 432-438****F****factorial algorithms, 254****factorial functions, creating, 254-255****features**

- of IDLE, 56-57
 - debugger, 56
 - help facility, 57
- of Raspberry Pi, 10-11

fifengine, 389**file extensions, .py, 63****File Manger icon (LXPanel), 38****file objects, 223-224****files**

- closing, 228-231
- managing, 220-221
- opening, 221-224

- preexisting files, writing to, 234
- reading, 224-228
 - entire file, 224-225
 - line by line, 225-227, 228-229
 - nonsequentially, 227-228
- types of, 217
- writing to, 231-235, 235-236
- find() function, 208**
- findall() function, 320**
- finditer() function, 320**
- flapping, 544**
- flavors of Python modules, 258**
- floating-point accuracy in calculations, 102**
- floating-point values, formatting, 212-213**
- floor division operator (//), 98**
- folders, present working directory, 31**
- Font object, creating, 399**
- for loops, 132-143**
 - assigning data types from a list, 135-136
 - iterating with variables, 137
 - iteration
 - with range function, 137-140
 - through character strings in a list, 136-137
 - through numbers in a list, 133-134
- nested loops, 149-150
- structure, 134-135
- traversing dictionaries with, 179
- validating user input with, 141-142
- format function, 102**
- format() function, 209-210**
 - positional formatting, 214-215
 - sign formatting, 213-214
- formatting**
 - long print lines with string concatenation, 79
 - numbers
 - floating-point values, 212-213
 - integer values, 212
 - output with print function, 73-75
 - strings for output, 209-215
 - format() function, 209-210
 - named placeholders, 210-211
 - positional placeholders, 210
 - webpage output, 475-476
- Fraction object, 103-105**
- frozensets, 187**
- fsum() function, 108**
- functions, 239-243**
 - append(), 164
 - array() function, 111
 - bind() function, 441
 - count() function, 166
 - creating, 240
 - defining, 240-243
 - dir(), 262
 - extend() function, 166
 - factorial functions, creating, 255-254
 - find() function, 208
 - findall() function, 320
 - finditer() function, 320
 - format function, 102
 - format() function, 209-210
 - fsum() function, 108
 - index() function, 167
 - input function, 88-89
 - int function, 89
 - len() function, 159
 - lists, 253-254
 - match() function, 319-320
 - modules, 257-259
 - built-in, 258
 - flavors of, 258
 - packages, 259
 - naming, 242-243
 - nested functions, 89
 - open() function, 221-222
 - passing values to, 244-250
 - arguments, 244-246
 - pop() function, 163
 - pow() function, 108
 - prcal(), 263
 - print function, 71-75
 - arguments, 72-73
 - output, formatting, 73-75
 - range function, 137-140
 - in re module, 319-320

- recursion, 254
- results, testing, 125-126
- reverse() function, 167
- reversed(), 169-170
- rfind() function, 208
- search() function, 320
- sort() function, 166
- sorted() function, 169-170, 179-180
- string-manipulation functions, 202-204
- type function, 87
- variables
 - global variables, 251-253
 - local variables, 251

G

- game designers, 388**
- game developers, 388**
- game frameworks, 388**
- game programming, 387-388**
 - game frameworks, 388
 - libraries, 388
 - PyGame library
 - adding sound to games, 407
 - event handling, 403-404
 - game screen, setting up, 397-398
 - installing, 390-391
 - modules, 394
- object classes, 396
- setting up, 388-394
- shapes, drawing, 404-407
- source code, obtaining, 392-393
- verifying installation, 390
- Raspberry Pie game, 410-417
- SDL, 388
- game screen**
 - displaying text on, 399-403
- graphics
 - interacting with, 409-410
 - moving, 408-409, 411-414
 - setting up, 397-398
- generic exception handling, 350**
- Gertboard**
 - circuit, building, 538-539
 - connecting to GPIO interface, 533-534
- get operation, 178-179**
- global variables, 251-253**
- GNOME, 34**
- GPIO (General Purpose Input/Output) interface, 529-534**
 - blinking the LED, 540-541
 - connecting to, 530-534
 - via Gertboard, 533-534
 - via Pi Cobbler, 532-533
- input
 - detecting, 542
 - events, 546-548
 - polling, 545-546
 - switch bounce, 548
- input signals, 544-545
- output
 - testing, 539-540
 - viewing, 536-539
- pin layout, 530
- PWM output, 530
- RPI.GPIO module, 535-536
 - installing, 535
 - startup methods, 535-536
- GPU (Graphics Processing Unit), 10**
- graphics**
 - interacting with, 409-410
 - moving on the game screen, 408-409, 411-414
- grouping**
 - comparisons with logical operators, 126-127
 - multiple statements
 - elif statement, 120-121
 - else statement, 118-120
 - if statement, 117-118
 - regular expressions, 331-332

GUI (graphical user interface), 29-43. See also GUI programming

- accessing, 29
- booting directly to, 35-36
- LXDE graphical interface, 35
 - desktop icons, 35
 - logging out of, 38-39
 - LXPanel, 36-39

GUI programming, 361-363, 381-382

- frame, 362
- packages, 363-364
 - tkinter package, 364-372
- widgets, 362
- window interface, 362

H

Hacking Raspberry Pi, 19

hash symbol (#), 80

HD (High Definition), 494-495

- megapixel rating, 494-495
- resolution, 494

HDMI cables, connecting Raspberry Pi to output display, 16

help function (modules), 261-262

help utility

- IDLE, 57
- PyGame library, 404
- Python interactive shell, 52

history of Raspberry Pi, 7-9

home directory, 31

HTML

- elements, parsing, 434-435
- files, serving, 471-472
- web forms, creating, 483-484

hyperbolic functions in math module, 108

I

icons

- File Manger icon (LXPanel), 38
- LXDE desktop icons, 35
- LXDE Programs Menu icon, 37-39

ICs (integrated circuits), SoC, 10

identifying

- maximum value in a tuple, 159-160
- minimum value in a tuple, 159-160
- number of values in a tuple, 159

IDLE, 55-57

- color coding, 56
- exiting, 57
- features, 56-57
 - debugger, 56
 - help facility, 57
- online documentation, 57

print function, 58-59

Python scripts

- creating, 62-63
- running, 60-62
- starting up, 55
- verifying installation, 47-48

if statement, 115-117

- condition checks, negating, 127-128
- multiple statements, grouping, 117-118
- syntax, 115

image handling, testing

PyGame library for, 405-407

image presentation script

- creating, 495-496
- images, locating, 497-498
- modifying, 513
- photos
 - centering, 503-507
 - framing, 503
 - scaling, 501-503
 - storing on removable drive, 498-501
- presentation screen, setting up, 496-497
- presentation speed, improving, 507-513

importing

- keywords into Python, 81-82
- math module to Python scripts, 106
- re module, 319

improving image presentation speed, 507-513

indentation

- in loops, 132
- multiple statements, grouping, 117-118

index() function, 167**index values (tuples), 157****infinite loops, 146-147****inheritance, 294-302,****307-313**

- subclasses, creating, 297-298
- syntax, 296

initializing PyGame library,**396-397****input, 91-90****GPIO**

- detecting, 542
- events, 546-548
- switch bounce, 548
- validating with for loops, 141-142

input function, 88-89**input polling, 545-546****input signals, GPIO, 544-545****installing**

- Apache web server, 470-471
- LXML module, 433
- module-building tools, 391-392
- MySQL, 448
- MySQL/Connector module, 452
- packages for PyGame library, 393-394
- PostgreSQL, 458
- psycopg2 module, 463

PyGame library, 390-391

Python, 48-50

RPI.GPIO module, 535

screen saver software package, 41-43

instantiating classes, 279**int function, 89****integers**

converting to strings, 89-90

formatting, 212

interacting with graphics on game screen, 409-410**interactive shell (Python),****51-54**

- exiting, 52
- exploring, 54
- help utility, 52
- Python scripts, running, 60
- Python v2, 51
- verifying, 46-47

interpreted programming languages, 50**isdigit() method, 125-126****iteration, 131**

- lists, 168-169
- for loop, 132-143
- for loops
 - iterating through character strings in a list, 136-137
 - iterating through numbers in a list, 133-134
 - iterating with range function, 137-140
 - iterating with variables, 137

tuples, 168-169

while loops, 143-148

using numeric

conditions, 143-144

using string conditions, 144-145

J-K**joining strings, 205****jumping to lines with syntactical errors (nano), 339****KDE, 34****keyboard**

- selecting, 16
- setup, verifying, 49-50

key/value pairs, 175

- accessing from dictionaries, 178
- dictionaries, populating, 176-178
- updating, 180-181

keywords, 81-82

- importing into Python, 81-82

kivy, 389**L****Label widget, 373****LAMP****(Linux-Apache-MySQL-PHP) server environment, 447****Leaf Pad, 64**

- LED, blinking, 540-541**
- len() function, 159**
- libraries for game programming, 388**
- LIFO (last in, first out), 164**
- line feeds, adding with escape sequence, 75-76**
- Linux, 27-28**
 - command line. *See* command line, 29-33
 - desktop environment, changing, 34
 - directory structure, 217-219
 - absolute directory references, 219
 - relative directory references, 219
 - distributions, 27
 - modular email environment, 425-427
- Linux shell, 29**
- list comprehensions, 170**
- Listbox widget, 378-380**
- listing functions in modules, 262**
- lists, 160-167. *See also* tuples**
 - concatenating, 165-166
 - creating, 160-161
 - extracting data from, 161-162
 - iteration, 168-169
 - list values
 - deleting, 163
 - popping, 163
 - replacing, 162
 - multidimensional lists, 167-168
 - creating, 167
 - referencing values in, 168
 - new data values, adding, 164-165
 - sorting, 169-170
 - using with functions, 253-254
- loading**
 - PyGame library, 396-397
 - Raspbian to SD card
 - from Linux, 556-558
 - from Mac OS, 558-560
 - from Windows, 553-556
- local variables, 251**
- locating images for presentation script, 497-498**
- logarithmic functions in math module, 108**
- logging in to Raspberry Pi, 28-29**
- logging out of Linux GUI sessions, 38-39**
- logical operators, 98-100**
 - comparisons, grouping, 126-127
- long print lines**
 - comments, adding, 80
 - formatting with string concatenation, 79
- loops**
 - indentation, 132
 - infinite loops, 146-147
 - for loop, 132-143
 - for loops
 - assigning data types from a list, 135-136
 - iterating through numbers in a list, 133-134
 - iterating with range function, 137-140
 - structure, 134-135
 - validating user input with, 141-142
 - nested loops, 149-150
 - while loops, 143-148
 - else clause, 145
 - entering data with, 147-148
 - iteration using numeric conditions, 143-144
 - iteration using string conditions, 144-145
 - terminating, 144
- ls command, 31**
- LXDE graphical interface, 35**
 - desktop icons, 35
 - LXPanel, 36-39
 - Digital Clock icon, 39
 - ScreenLock icon, 39
- LXDE Logout Manager, 38-39**
- LXDE Programs Menu icon, 37-39**
- LXML module**
 - CSSSelector() method, 435-436
 - etree methods, 434-435
 - installing, 433

LXPanel, 36-39

- Digital Clock icon, 39
- File Manger icon, 38
- LXDE Logout Manager, 38-39
- LXDE Programs Menu icon, 37-39
- ScreenLock icon, 39

LXTerminal icon (LXDE), 35**M****managing**

- dictionaries, 181-182
- directories, 220-221

manipulating

- files, 220-221
- strings with functions, 202-204
 - string-joining functions, 205
 - string-splitting functions, 204-205
 - string-testing functions, 205-206

mappings, 178**match() function, 319-320****math module, 106-107. See also NumPy module**

- hyperbolic functions, 108
- logarithmic functions, 108
- number theory functions, 107-108

statistical math functions, 110

trigonometric functions, 108

mathematical operators, 97-100

- binary operators, 98
- floor division operator, 98
- logical operators, 98-100
 - comparisons, grouping, 126-127
 - order of operations, 100
 - shortcuts, 103

maximum value in a tuple, identifying, 159-160**megapixel rating, 494-495****Menu widget, 380-381****methods, 280-287**

- accessor methods, 281-283
- constructors, 283-284
- CSSSelector() method, 435-436
- ehlo() method, 429
- file object methods, 223-224
- isdigit() method, 125-126
- mutator methods, 280-281
- os methods, 220-221
- overriding, 306
- property() method, 286-287
- range() method, 171
 - in socket module, 439

minimum value in a tuple, identifying, 159-160**mkdir command, 31****Model A, 7**

features, 10-11

Model B, 10

features, 10-11

modifying

- image presentation script, 513
- sets, 192-193

modules, 257-259

- available modules, exploring, 264-265
- built-in, 258
- cgi module, 484-487
- class modules, creating, 287-289
- custom modules
 - creating, 264-272, 272-273
 - testing, 267-268, 272
- flavors of, 258
- help function, 261-262
- installing tools for building, 391-392
- listing functions in, 262
- moving to a production directory, 268-270
- MySQL/Connector module, installing, 452
- naming, 266
- networking modules, 423
- packages, 259
- psycopg2 module
 - coding with, 463-466
 - installing, 463

- in PyGame library, 394
- re module, 319-321
 - compiled regular expressions, 320-321
- RPI.GPIO module, 535-536
 - installing, 535
 - startup methods, 535-536
- smtplib module, 427-429
- socket module, 439
- standard modules, 259-261

moving

- graphics on the game screen, 408-409
- modules to a production directory, 268-270
- operating system to SD card, 21

multidimensional lists, 167-168

- creating, 167
- referencing values in, 168

multiple statements, grouping

- elif statement, 120-121
- else statement, 118-120
- if statement, 117-118

music

- playback, controlling, 517-521
- playing, 514-521
- playlists, 516-517
- special presentations, 521-525
- storing on removable drive, 515-516

mutator methods, 280-281

MySQL, 447-458

- databases
 - connecting to, 452
 - creating, 449-450
 - inserting data, 455-457
 - querying data, 457-458
- environment, setting up, 448-452
- installing, 448
- tables, creating, 451-452
- user accounts, creating, 450-451

MySQL/Connector module, installing, 452

N

named placeholders, 210-211

naming

- functions, 242-243
- modules, 266
- variables, 82

nano

- commands, 65
- jumping to lines with syntactical errors, 339
- Python scripts, creating, 64-65
- verifying installation, 48
- website, 65

negating

- character classes, 327-328
- condition checks, 127-128

nested functions, 89

nested loops, 149-150

network cables, 18-19

networking, email servers, 423-432

networking modules, 423

- urllib module, 432-433

number theory functions in math module, 107-108

number values, assigning to variables, 85-86

numbers

- displaying, 102-103
- floating-point values, formatting, 212-213
- integer values, formatting, 212

numeric comparison operators, 122

NumPy module, 110-112

- arrays, 111-112
- data types, 110-111

O

obtaining

- information from sets, 189-192
 - set difference, 190-191
 - set intersection, 190
 - set membership, 189
 - set union, 189-190
 - symmetric set difference, 191
- source code for PyGame library, 392-393

official Python website, 57

online documentation,
IDLE, 57

OOP (object-oriented programming), 277-280

- classes, 278-279
 - class modules, creating, 287-289
 - the class problem, 293-294
 - inheritance, 295
 - instantiating, 279
 - methods, 280-287
 - subclasses, 294-295
- default attribute values, 279-280

open() function, 221-222

- mode, designating, 222-223

opening files, 221-224

operating systems

- downloading, 21
- image, troubleshooting, 25
- Linux, 27-28
 - desktop environment, changing, 34
 - modular email environment, 425-427
- moving to SD card, 21
- Raspbian
 - loading to SD card, 553-560
 - updating, 41-43
- selecting, 20

operators, mathematical operators, 97-100

- binary operators, 98
- logical operators, 98-100
- order of operations, 100
- shortcuts, 103

OR operators, 98

order of operations, changing, 100

os methods, 220-221

output, 91-90

- { } placeholder, 102-103
- of classes, customizing, 284-285
- controlling with escape sequences, 75-76
- formatting with print function, 73-75
- GPIO
 - testing, 539-540
 - viewing, 536-539
- producing from Python scripts, print function, 71-78
- separators, 83
- strings, formatting, 209-215
 - format() function, 209-210
 - positional placeholders, 210

output display, selecting, 15-16

overriding a method, 306

P

packages, 259

- in GUI programming, 363-364
 - tkinter package, 364-372
- for PyGame library, installing, 393-394
- PyPi, 261

Panda3D, 389

parameters, 244-246

- default values, setting, 246-247
- multiple parameters, retrieving, 248-249
- positional parameters, 246

parsing

- data from webpages, 433
- HTML elements, 434-435

passing by reference, 253

passing values to functions, 244-250

- arguments, 244-246

password prompt, 32

passwords, changing, 32-33

path directories, checking, 270-271

patterns for regular expressions

- anchor characters, 324-325
- asterisk, 329
- braces, 330-331

- character classes, 326-327
- dot character, 325-326
- grouping, 331-332
- pipe symbol, 331
- plain text, 321-323
- plus sign, 330
- question mark, 330
- ranges, 328-329
- special characters, 323
- performing repetitive tasks, 131**
- peripherals**
 - cases, 17-18
 - cords, troubleshooting, 24
 - network cables, 18-19
 - plugging in, 21-23
 - purchasing, 19
 - required peripherals, 11-16
 - keyboard, 16
 - output display, 15-16
 - power supply, 13-15
 - SD card, 12-13
 - SD card, transferring operating system to, 21
 - self-powered USB hub, 18
 - USB mouse, 18
 - verifying operation, 21-23
- phone number validator script, 333-334**
- photos**
 - centering, 503-507
 - scaling, 501-503
- special presentations, 521-525
- storing on removable drive, 498-501
- Pi Cobbler**
 - circuit, building, 537-538
 - connecting to GPIO interface, 532-533
- pin layout, GPIO interface, 530**
- pipe symbol (|), 331**
- placeholders**
 - { } placeholder, 102-103
 - named placeholders, 210-211
 - positional placeholders, 210
- plain text as regular expression pattern, 321-323**
- playing music, 514-521**
 - playlists, 516-517
 - songs, queuing, 515
- playlists, 516-517**
 - randomizing, 521
- plugging in peripherals, 21-23**
- plus sign (+), 330**
- polling, 545-546**
- polymorphism, 306**
- pop() function, 163**
- popping list values, 163**
- populating**
 - dictionaries, 176-178
 - sets, 188-189, 194
- portable power supplies, 15**
- positional formatting, 214-215**
- positional parameters, 246**
- positional placeholders, 210**
- POSIX BRE (Basic Regular Expression) engine, 318**
- POSIX ERE (Extended Regular Expression) engine, 318**
- PostgreSQL, 447, 458-466**
 - databases
 - connecting to, 463-464
 - creating, 459-460
 - querying data, 465-466
 - environment, setting up, 459-463
 - inserting data, 464-465
 - installing, 458
 - tables, creating, 461-463
 - user accounts, creating, 460-461
- pow() function, 108**
- power supply**
 - portable power supplies, 15
 - selecting, 13-15
- prcal() function, 263**
- preexisting files, writing to, 234**
- prepackaged kits, 11-12**
- present working directory, 31**
- print function, 71-75**
 - arguments, 72-73
 - IDLE, 58-59
- private attributes, 281**
- privileges**
 - root account, 32
 - sudo command, 32

procedural programming, 277
programming

- with dictionaries, 182-187
- event-driven programming, 362-363
- game programming. *See* game programming, 381-382
- GUI programming, 361-363, 381-382
 - tkinter package, 363-364
 - widgets, 362
 - window interface, 362
- OOP, 277-280
 - classes, 278-279
- procedural programming, 277
- with sets, 193-196
- socket programming, 438-445
- web programming, 447
- property() method, 286-287**
- protecting quotes with escape sequences, 76**
- psycogp2 module**
 - coding with, 463-466
 - installing, 463
- publishing**
 - database data on the web, 478
 - webpages, 472
- purchasing Raspberry Pi, 9-11**
 - required peripherals, 11-16
 - keyboard, 16
 - output display, 15-16

- power supply, 13-15
- SD card, 12-13

pwd command, 31**PWM output, 530, 541-542****.py file extension, 63****PyGame library, 389**

- event handling, 403-404
- game screen
 - displaying text on, 399-403
 - setting up, 397-398
- graphics
 - interacting with
 - on game screen, 409-410
 - moving on the game screen, 408-409, 411-414

help facility, 404

image handling, testing, 405-407

initializing, 396-397

installing, 390-391, 394

loading, 396-397

modules, 394

object classes, 396

packages, installing, 393-394

setting up, 388-394

shapes, drawing, 404-407

sound, adding to games, 407

source code, obtaining, 392-393

verifying installation, 390

Pyglet, 389**PyPi (Python Package Index), 261****PySoy, 389****Python, 9. *See also* Python scripts**

- development environment
 - shell, 50, 53-57. *See also* IDLE
 - syntax, color coding, 53
- history of, 45-46
- installing, 48-50
- interactive shell, 51-54
 - exiting, 52
 - exploring, 54
 - help utility, 52
- official Python website, 57
- reasons for learning, 45
- scripts, 51
- versions, comparing, 46

Python interpreter, verifying, 46-47**Python scripts, 57-65**

- comments, adding, 80
- creating
 - commands used for, 65
 - in IDLE, 62-63
 - with nano, 64-65
- functions, 240-243
- image presentation script
 - creating, 495-496
- images, locating, 497-498
- presentation screen,
 - setting up, 496-497

inheritance, 299-302
input, 90-91
input, allowing, 88-89
keywords, 81-82
music script, creating, 514-515
output, 90-91
 producing with print function, 71-78
phone number validator, 333-334
Raspberry Pi game, 410-417
running
 commands used for, 65
 in IDLE, 60-62
 in interactive shell, 60
separators, 83
users, 79
variables
 assigning value to, 82-86
 data types, 86-88
 long string values, assigning, 84-85
 naming, 82
 number values, assigning, 85-86
 reassigning values to, 86
 unassigned, 84
Python v2, interactive shell, 51
Python-Ogre, 389

PYTHONPATH environment variable, 271

Q

querying data
 with MySQL, 457-458
 with PostgreSQL, 465-466
question mark (?), 330
queuing songs, 515
quotes, protecting with escape sequences, 76

R

randomizing playlists, 521
range function, 137-140
range() method, 171
ranges, 170-171
 as pattern for regular expressions, 328-329
Raspberry Pi
 acquiring, 9-11
 features, 10-11
 history of, 7-9
 logging in to, 28-29
 prepackaged kits, 11-12
 troubleshooting, 24-25
Raspberry Pi Foundation, 9, 19
Raspberry Pie game, 410-417

Raspbian

loading to SD card
 from Linux, 556-558
 from Mac OS, 558-560
 from Windows, 553-556
updating, 41-43

re module

compiled regular expressions, 320-321
functions, 319-320
importing, 319

reading files, 224-228

entire file, 224-225
line by line, 225-227, 228-229
nonsequentially, 227-228

reasons for learning Python, 45

reassigning values to variables, 86

reboot command, 31

rebooting, 32

recursion, using with functions, 254

referencing

strings, 201-202
values in multidimensional lists, 168

regular expressions, 317-318

advanced features, 330-332

grouping, 331-332

patterns

 anchor characters, 324-325

- asterisk, 329
- braces, 330-331
- character classes, 326-327
- dot character, 325-326
- pipe symbol, 331
- plain text, 321-323
- plus sign, 330
- question mark, 330
- ranges, 328-329
- special characters, 323
- phone number validator script, 333-334
- POSIX BRE engine, 318
- POSIX ERE engine, 318
- re module, 319-321
 - compiled regular expressions, 320-321
 - functions, 319-320
- relational databases, 451**
- relative directory references, 219**
- remote email servers, 428**
- remove operation, 193**
- repetitive tasks, performing, 131**
- replacing list values, 162**
- required peripherals, 11-16**
 - keyboard, 16
 - output display, 15-16
 - power supply, 13-15
 - SD card, 12-13
 - transferring operating system to, 21

- resolution, 494**
 - megapixel rating, 494-495
- results of expressions, assigning to variables, 85-86**
- results of functions, testing, 125-126**
- retrieving**
 - data from websites, 437-438
 - multiple parameters, 248-249
 - values with dictionaries, 249-250
 - webpages, 432-433
- return statement, 243-244**
- reverse() function, 167**
- reversed() function, 169-170**
- reviewing, commands, 54**
- RFC 2822 email standard, 429**
- rfind() function, 208**
- root account, 32**
- RPI.GPIO module, 535-536**
 - installing, 535
 - startup methods, 535-536
- running**
 - client programs, 443-445
 - Python scripts
 - commands used for, 65
 - in IDLE, 60-62
 - in interactive shell, 60
 - server programs, 443-445
- runtime error exceptions, 340-342**

S

- scaling photos, 501-503**
- screen saver software package**
 - installing, 41-43
- ScreenLock icon (LXPanel), 39**
- scripts (Python), 51, 57-65**
- SD card**
 - loading Raspbian to
 - from Linux, 556-558
 - from Mac OS, 558-560
 - from Windows, 553-556
 - selecting, 12-13
 - transferring operating system to, 21
 - troubleshooting, 24
- SDL (Simple DirectMedia Layer), 388**
- search() function, 320**
- searching strings, 207-209**
- selecting**
 - cases, 17-18
 - keyboard, 16
 - operating system, 20
 - output display, 15-16
 - power supply, 13-15
 - SD card, 12-13
 - Wi-Fi adapters, 18-19
- self-powered USB hub, selecting, 18**
- sending email messages, 430-432**
- separators, 83**

- server programs, 438-439**
 - creating, 439-441
 - running, 443-445
- serving HTML files, 471-472**
- set difference, 190-191**
- set intersection, 190**
- set theory, 189**
- set union, 189-190**
- sets, 187**
 - creating, 187-188
 - elements, membership, 189
 - frozensets, 187
 - modifying, 192-193
 - obtaining information from, 189-192
 - populating, 188-189, 194
 - programming with, 193-196
 - traversing, 191-192
- shallow copy, 184**
- shapes, drawing with PyGame library, 404-407**
- shortcuts for mathematical operators, 103**
- shutting down Raspberry Pi, 23**
- sign formatting (format() function), 213-214**
- single quotes ('), print function syntax, 72-73**
- slicing, 202**
- smtplib module, 425, 427-429**
- SoC (system on a chip), 10**
- socket module, 439**
- socket programming, 438-445**
 - client programs, 438-439
 - creating, 441-444
 - running, 443-445
 - server programs
 - creating, 439-441
 - running, 443-445
 - servers, 438-439
- songs**
 - playback, controlling, 517-521
 - playing continuously, 522-525
 - playlists, 516-517
 - randomizing, 521
 - queuing, 515
 - special presentations, 521-525
- sort() function, 166**
- sorted() function, 169-170, 179-180**
- sorting lists, 169-170**
- sound, adding to games, 407**
- source code, obtaining for PyGame library, 392-393**
- special characters as regular expression pattern, 323**
- speed of presentations, improving, 507-513**
- splitting strings, 204-205**
- sprites, 396**
- stacks, 164**
- standard modules, 259-261**
- starting up IDLE, 55**
- statements**
 - commands for testing, 65
 - try except statement, 344
 - creating multiple blocks of, 347-350
 - options, 350-352
- statistical math functions in math module, 110**
- storing**
 - music on removable drive, 515-516
 - photos on removable drive, 498-501
 - subclasses in its own object module file, 301-302
- string comparison operators, 122-124**
- string concatenation, 79**
- string literals, 72**
- strings**
 - ASCII code, 200
 - converting to integers, 89-90
 - creating, 200-201
 - formatting
 - positional formatting, 214-215
 - sign formatting, 213-214
 - formatting for output, 209-215
 - format() function, 209-210
 - named placeholders, 210-211
 - positional placeholders, 210

- joining, 205
- manipulating with
 - functions, 202-204
- referencing, 201-202
- regular expressions, 317-318
 - advanced features, 330-332
 - POSIX BRE engine, 318
 - POSIX ERE engine, 318
 - re module, 319-321
- searching, 207-209
- slicing, 202
- splitting, 204-205
- testing, 205-206
- structure of for loops, 134-135**
- structured commands, 115**
 - if statement, 115-117
 - multiple statements, grouping, 117-118
 - syntax, 115
- subclasses, 294-295, 308-314**
 - adding to object module file, 298-300
 - creating, 297-298
 - polymorphism, 306
 - storing in its own object module file, 301-302
- sudo command, 32**
- superclasses, 294**
- superuser, 32**
- switch bounce, 548**
- symmetric set difference, 191**

- synchronous events, 546-547**
- syntactical error exceptions, 337-339**
- syntax**
 - color coding, 53
 - if statement, 115
 - for inheritance, 296
 - input function, 88
 - for loop, 132
 - print function, 73-75
 - while loops, 143

T

- tables, creating**
 - for MySQL databases, 451-452
 - for PostgreSQL databases, 461-463
- televisions, connecting Raspberry Pi to analog televisions, 15**
- terminating**
 - infinite loops, 146-147
 - while loops, 144
- test directories, creating modules in, 266-267**
- testing**
 - custom modules, 267-268, 272
 - function results, 125-126
 - GPIO output, 539-540
 - Python statements, commands used for, 65
 - strings, 205-206
- text editors**
 - Leaf Pad, 64
 - nano
 - commands, 65
 - jumping to lines with syntactical errors, 339
 - Python scripts, creating, 64-65
 - verifying installation, 48
- Text widget, 377-378**
- tkinter package, 364-372**
 - Button widget, 373
 - Checkbutton widget, 373-375
 - Entry widget, 397-398
 - event handlers, defining, 370-372
 - Label widget, 373
 - Listbox widget, 378-380
 - Menu widget, 380-381
 - Text widget, 377-378
 - windows, creating, 364-366
- transferring operating system to SD card, 21**
- transparency, adding to images, 405-406**
- traversing sets, 191-192**
- trigonometric functions in math module, 108**
- triple quotes, 74-78**
- troubleshooting Raspberry Pi, 24-25**
 - CGI, 480-482
 - peripheral cords, 24

try except statement, 344
 creating multiple blocks of, 347-350
 options, 350-352

tuples, 155-160
 accessing data in, 157
 comparing, 124
 comparison operations, 158-159
 concatenating, 160
 creating, 155-156
 index values, 157
 iteration, 168-169
 maximum value, identifying, 159-160
 minimum value, identifying, 159-160
 number of values in, identifying, 159
 range of values, accessing, 157-158

turning off Raspberry Pi, 23

type function, 87

U

unassigned variables, 84

Unicode, 46
 displaying, 76-77

updating
 dictionaries, 180-181
 Raspbian Linux distribution software, 41-43
 sets, 192-193

upside-down trees, 217

Upton, Eben, 7

U.S. keyboard, configuring, 49-50

USB mouse, selecting, 18

user accounts
 MySQL, creating, 450-451
 PostgreSQL, creating, 460-461

users, 79

utilities, Linux shell, 29

V

validating
 custom modules, 272
 GPIO output, 539-540
 user input with for loops, 141-142

values
 comparing
 Boolean comparisons, 124-125
 numeric comparisons, 122
 string comparisons, 122-124
 passing to functions, 244-250
 arguments, 244-246
 retrieving with dictionaries, 249-250
 returning, 243-244

van Rossum, Guido, 45

variables, 81-82. See also object references
 assigning value to, 82-86
 long string values, 84-85
 number values, 85-86
 calculation results, assigning, 101
 camel Case, 194
 data types, 86-88
 converting, 89-90
 in equations, 100-101
 global variables, 251-253
 local variables, 251
 naming, 82
 parameters, 244-246
 default values, setting, 246-247
 multiple parameters, retrieving, 248-249
 positional parameters, 246
 reassigning values to, 86
 separators, 83
 unassigned, 84

verifying
 IDLE, 47-48
 keyboard setup, 49-50
 peripherals, 21-23
 PyGame library installation, 390
 Python interactive shell, 46-47
 Python interpreter, 46-47
 text editor installation, 48

versions of Python,
comparing, 46
viewing GPIO output, 536-539

W

web forms, 482-487
creating, 483-484
web programming, 447
CGI, 473-475
dynamic webpages,
476-480
serving HTML files,
471-472
web forms, 482-487
creating, 483-484
webpages, formatting
output, 475-476
web servers
Apache web server,
473-475
installing, 470-471
HTML elements, parsing,
434-435
webpages
parsing data from, 433
retrieving, 432-433

webpages
CGI, debugging, 480-482
dynamic webpages,
476-480
output, formatting,
475-476
publishing, 472
retrieving, 432-433

websites
nano editor homepage, 65
official Python website, 57
PyGame wiki, 394
PyPi, 261
Raspberry Pi Foundation,
19
retrieving data from,
437-438

while loops, 143-148
else clause, 145
entering data with,
147-148
iteration
using numeric
conditions, 143-144
using string conditions,
144-145
nested loops, 149-150
terminating, 144
whoami command, 29

widgets, 362
adding to windows,
366-370
Button widget, 373
Checkbutton widget,
373-375
Entry widget, 397-398
Label widget, 373
Listbox widget, 378-380
Menu widget, 380-381
**Wi-Fi adapters, selecting,
18-19**
**window interface (GUI
programming), 362**
windows
creating, 364-366
widgets, adding, 366-370
writing to files, 231-236
preexisting files, 234

X-Y-Z

Xfce, 34