Richard Blum
Christine Bresnahan

Sams Teach Yourself

Python Programming for Raspberry Pi

in 24 Hours

SAMS

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

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

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

Part I: The Raspberry Pi Programming Environment

Part II: Python Fundamentals

Part III: Advanced Python

Part IV: Graphical Programming

Part V: Business Programming

Part VI: Raspberry Pi Python Projects

Part VII: Appendix

Table of Contents

Part I: The Raspberry Pi Programming Environment

Part II: Python Fundamentals

Part III: Advanced Python

Part IV: Graphical Programming

Part V: Business Programming

Part VI: Raspberry Pi Python Projects1

Part VII: Appendix

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 `monospace` font.
Commands	Commands and their syntax use **bold**.
Menu commands	We use the following style for all application menu commands: *Menu*, *Command*, where *Menu* is the name of the menu you pull down and *Command* 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

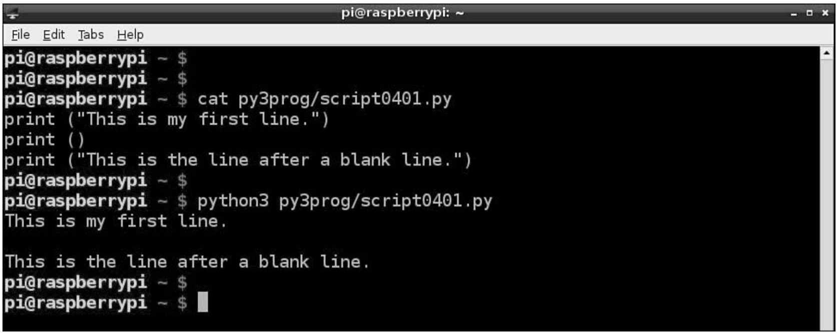
You can also embed single quotes within single quote marks and double quotes within double quote marks. However, when you do, you need to use something called an "escape sequence," which is covered later in this hour.

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 π!
>>>
```

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!
>>>
```

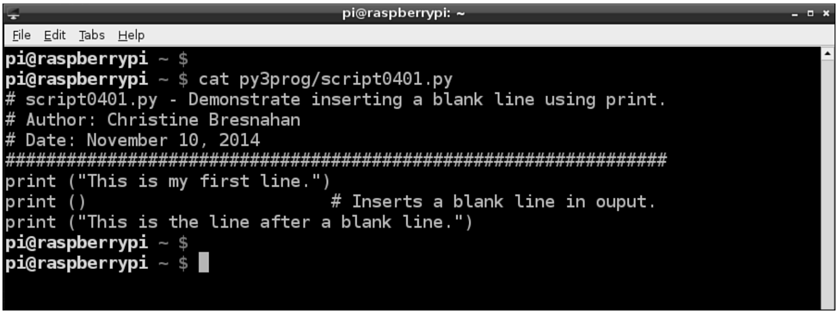
It is always a good rule to keep your Python syntax simple to provide better readability of the scripts. However, sometimes you need to use complex syntax. This is where comments will help you. No, not comments spoken aloud, like “I think this syntax is complicated!” We’re talking about comments that 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 ()                        # Inserts a blank line in ouput.
print ("This is the line after a blank line.")
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
`float`	Floating-point number
`int`	Integer
`long`	Long integer
`str`	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 #################
#
```

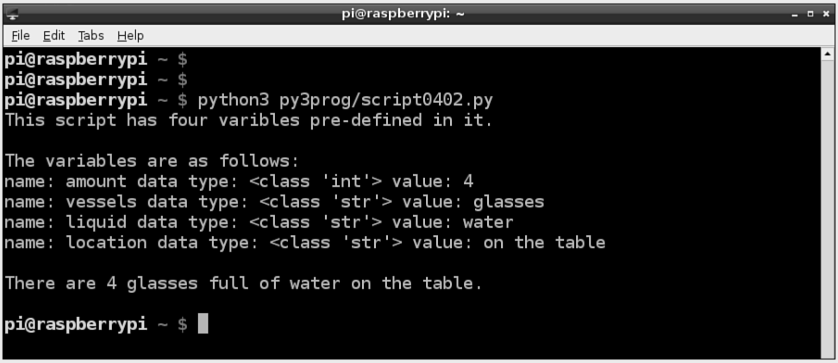
```
print ()
#
print ("The variables are as follows:")
#
print ("name: amount", "data type:", type (amount), "value:", amount)
#
print ("name: vessels", "data type:", type (vessels), "value:", vessels)
#
print ("name: liquid", "data type:", type (liquid), "value:", liquid)
#
print ("name: location", "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 varibles 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.

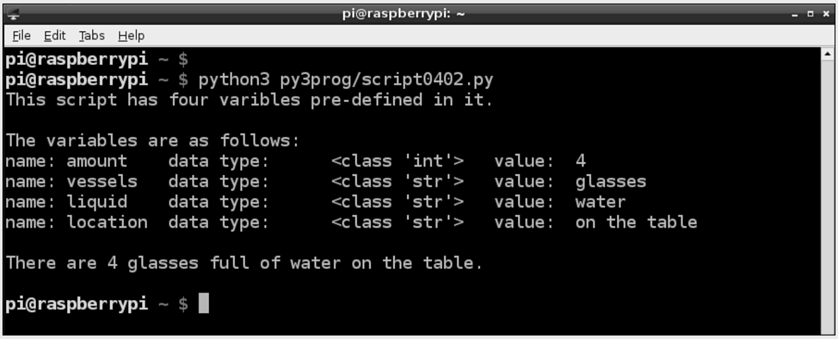
10. Go to the `Output Variable Description` portion of the script and add a separator to the end of each line. The lines of code to be changed and how they should look when you are done are shown here:

```
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 varibles 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.

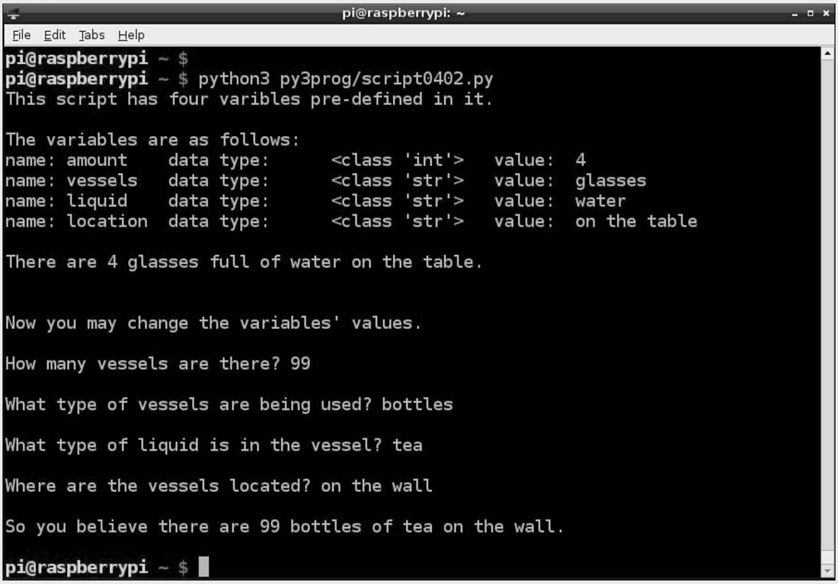
15. Go to the bottom of the script and add the additional Python statements shown here:

```
#
################## 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 ~ $
pi@raspberrypi ~ $ python3 py3prog/script0402.py
This script has four varibles 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