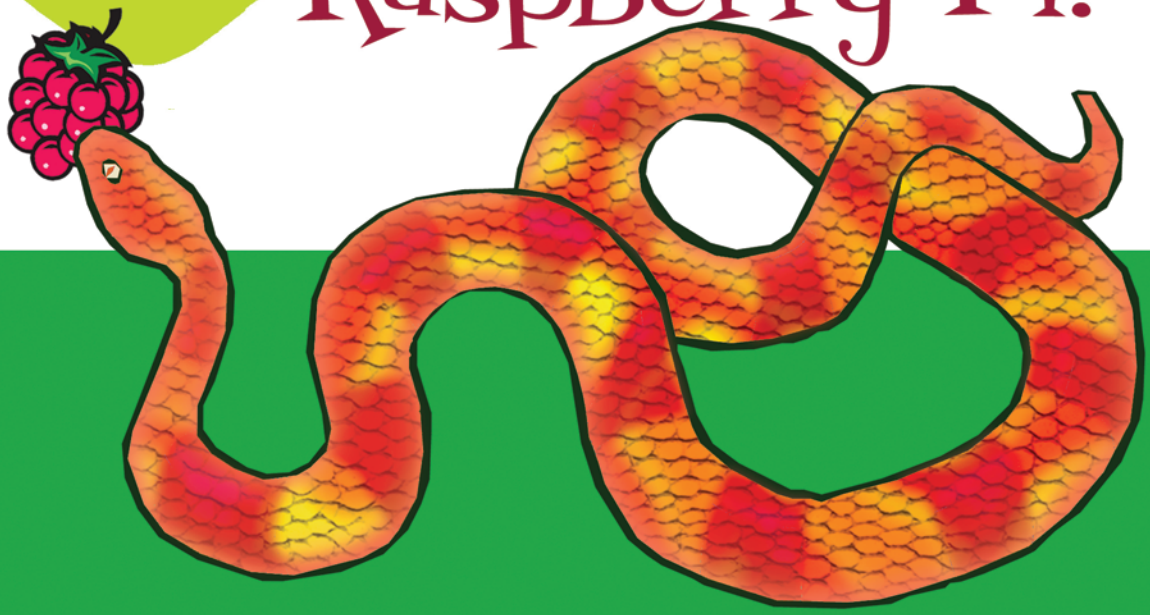
Python programming for kids and other beginners

Hello Raspberry Pi!™



Ryan Heitz

MANNING

Hello Raspberry Pi!

Python programming for kids and other beginners

Ryan Heitz

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2016 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without elemental chlorine.

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Dan Maharry
Copyeditor: Tiffany Taylor
Proofreader: Alyson Brener
Technical proofreader: Romin Irani
Typesetter: Marija Tudor
Cover designer: Leslie Haimes

ISBN: 9781617292453

Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 18 17 16 15 14 13

To Juliana, Daniel, and John

Brief contents

PART 1 GETTING STARTED 1

1 Meet Raspberry Pi 3

2 Exploring Python 33

PART 2 PLAYING WITH PYTHON 65

3 Silly Sentence Generator 3000: creating interactive programs 67

4 Norwegian Blue parrot game: adding logic to programs 91

5 Raspi's Cave Adventure 121

PART 3 PI AND PYTHON PROJECTS 149

6 Blinky Pi 151

7 Light Up Guessing Game 176

8 DJ Raspi 204

Contents

Preface xiii
Acknowledgments xv
About this book xvii

PART 1 GETTING STARTED 1

1 Meet Raspberry Pi 3

What is the Raspberry Pi? 4

Exploring your Raspberry Pi's parts: hardware 4

Giving your Pi a cozy home: Pi cases 6 ❍ The brain of your Pi: system on a chip 7 ❍ Connecting a keyboard and mouse: USB ports 8 ❍ Storing memories: your Pi gets a memory card 10 Connecting a TV or monitor: HDMI port 13 ❍ Other ports and connections 17 ❍ Powering your Pi: microUSB power port 17 It's alive! Plugging in the Pi 18

Getting your Pi running: software 19

Installing the Raspbian operating system 19 ❍ Configuring the operating system: making it yours 21 ❍ Saving your configuration and rebooting 24

Getting around: learning Raspbian 26

Finding and opening applications on your Raspberry Pi 26 Your files and folders 26 ❍ Writing code 28

Fruit Picker Extra: shopping at the Pi Store 29

Challenge 30
Scavenger hunt 31

Summary 31

2 Exploring Python 33

Playing with Python 33

Discovering Python's mathematical operators 35
Adding and subtracting 35 ❍ Multiplying and dividing 37
Figuring out whole numbers and remainders 38
Exponents 38 ❍ Square roots 39 ❍ Challenge: stacking Pis! 39

Storing information using variables 41
Creating variables and assigning values 42 ❍ Displaying variable values 42 ❍ Storing strings in variables 45
Changing the value of variables 46

Displaying text on a screen 50
Using the print function 50 ❍ Troubleshooting 51

Creating programs 52
Writing Python programs with IDLE 53 ❍ Starting a new program 54 ❍ Saving programs 56 ❍ Python interpreting the program 57

Fruit Picker Extra: creating documents 57
Writing silly things and saving them 57

Challenges 60
The matrix 61 ❍ Building a brick wall 61
Pi electrons 62

Summary 62

PART 2 PLAYING WITH PYTHON 65

3 Silly Sentence Generator 3000: creating interactive programs 67

Creating a welcome message 68
Starting a new program 69 ❍ Saving the program 71

Adding notes in your code 73
Using hashtags for comments 73

Getting and storing information 75

Joining strings 77
Using more than one input 79 ❍ Building the sentence 80
Troubleshooting 81

Completing the program: displaying the silly sentence 83

Fruit Picker Extra: Minecraft Pi 85
What's Minecraft? 85 ❍ Launching Minecraft Pi 86
Python programming interface to Minecraft Pi 88

Challenges 88
Knight's Tale Creator 3000 88 ❍ Subliminal messages 89

Summary 90

4 Norwegian Blue parrot game: adding logic to programs 91

Displaying the game introduction 92
Creating the game welcome message and instructions 94

Collecting input from the player 101

Using if statements to respond to users in different ways 105
Practicing if statements 108

Using while loops to repeat things 110
A closer look at while loops 112 ❍ Breaking out of a while loop 113 ❍ Practicing while loops 114

Using Python code libraries to generate random numbers 115

Fruit Picker Extra: Scratch 118

Challenges 119

Summary 120

5 Raspi's Cave Adventure 121

Project introduction: Raspi's Cave Adventure 122
Left cave 124 ❍ Right cave 124

Hey wait, you need a plan (flow diagrams) 124

Which way should Raspi go? (checking input) 126

Handling unexpected input 127 ❍ Turning flow diagrams into code 131

Simplify! Making your own functions 133

Finishing the left cave 138 ❍ Exploring the right cave 139 Troubleshooting 141

Fruit Picker Extra: playing video 142

Live streaming: exploring from your Pi 143

Challenges 145

Introducing dramatic pauses 145 ❍ Random demise 146 Play again? 147 ❍ Scream! 147

Summary 147

PART 3 PI AND PYTHON PROJECTS 149

6 Blinky Pi 151

Setting up your Pi for physical computing 153

GPIO pins 153 ❍ Breaking out the GPIO pins to a breadboard 155 ❍ Breadboard basics 158

Building the LED circuit 161

Step 1. Connect the jumper from GPIO pin 21 163 Step 2. Add the red LED 164 ❍ Step 3. Connect a resistor 164

Software: blinkLED program 166

Running the program 168 ❍ blinkLED: how it works 169

Adding more LEDs 171

Building the circuit 171

Multiple LEDs: program it! 173

Challenges 174

Wave pattern 174 ❍ Simon Says 174 ❍ Random blinking 174

Summary 175

7 Light Up Guessing Game 176

Guessing Game design 178

Hardware: building the circuit 179

Numbers, numbers, numbers! 179 ❍ Wiring an RGB LED 180 ❍ Circuit sketch 180

Software: LEDGuessingGame program 188

Setting up the GPIO pins for the RGB LED 190 ❍ Main game loop and logic 195 ❍ Guessing Game Loop and logic 197 Adding the Play Again Loop and logic 198 ❍ Playing the game 200 ❍ Troubleshooting 200

Challenges 201

Game winner 201 ❍ Easter egg 201 ❍ Warmer and colder 201 ❍ Darth Vader surprise 202

Summary 202

8 DJ Raspi 204

Project overview 205

Setting up your Pi to play sounds 207

OMXPlayer and MP3s 208 ❍ Troubleshooting 209

Hardware: building the circuit 210

Wiring a button 210 ❍ Circuit sketch 211 ❍ Adding the second button 217

Software: the DJ Raspi program 218

Setting up the Pi: initializing the buttons 220 ❍ Getting a list of sounds 221 ❍ Getting a value of an item stored in a list 225 ❍ Getting the length of a list 226 ❍ Building a list of sound files with the os library 227 ❍ Playing a sound when a button is pressed 228 ❍ Functions! 231 Testing: your first gig as DJ Raspi 234

Troubleshooting 235

Challenges 236

Double button press surprise 236 ❍ Yoda Magic 8 Ball 236 ❍ Continuing to explore 237

Summary 237

Appendix A Raspberry Pi troubleshooting 239
Appendix B Raspberry Pi ports and legacy boards 245
Appendix C Solutions to chapter challenges 261
Appendix D Raspberry Pi projects 279

Index 285

Preface

In 2013, a parent and friend of mine asked if I would teach a Python course to middle school students at a local school. My friend gently asked if I could somehow use the Raspberry Pi computer in the course. I love learning new things and I had been reading a lot about the Raspberry Pi. So as you can imagine, I was tremendously excited at the opportunity of using it and emphatically said "Yes!" That event began my journey of developing a course for kids on programming in Python and using the Raspberry Pi and later, this book.

Quickly, as I worked with the Raspberry Pi, I became a disciple of the Raspberry Pi inventors: the best way for kids to learn programming is by giving them an affordable, ready-to-program computer. It was the perfect platform to learn how to program.

As a teacher of computer science, I grew to deeply appreciate Python. I became convinced that it was not only a great programming language, but its focus on readability and simplicity made it perfect for kids to learn as their first programming language.

Fast forward in time—after teaching Python using the Raspberry Pi to many classes of kids, I had developed a set of engaging and funny projects that the kids enjoyed. Just as important, the students learned! The feedback from the kids and the parents was fantastic! Imagine kids rushing to take part in a programming class. It was wonderful!

A few months after developing my course, Nicole Butterfield and Robin de Jongh of Manning Publications contacted me about turning it into a

book. I was thrilled at the prospect of bringing the activities and projects from the computer lab into the hands of kids everywhere. What is more, this book would fill an important gap. What I had found when I originally started teaching my course was that there were no books on the Raspberry Pi and programming in Python that were designed for kids. Since the main reason for inventing the Raspberry Pi was to get more kids programming, I was enthusiastic to work on this project.

Nearly two years later, and several versions of the Raspberry Pi later, I'm proud to present this book to the kids and other beginners who want to learn to program. I hope you enjoy using this book and it starts you on your own journey in computer science!

Acknowledgments

Thank you to my wife, Juliana, and our two children, Daniel and John, for their endless support and patience through the long days, nights, and weekends I needed to write this book.

I'd also like to thank Manning Publications for having the vision to pursue this project. In particular, thanks to Robin de Jongh and Nicole Butterfield who kicked off this project by finding and encouraging me; to publisher Marjan Bace for his commitment to me and to this book; to Ozren Harlovic for orchestrating the book review process; to Kevin Sullivan and Mary Piergies for overseeing production; to Chuck Larson for the wonderful work on the graphics; to Tiffany Taylor for her outstanding copyediting; to Alyson Brener for her thorough proofreading; to Candace Gillhoolley and Ana Romac for promoting the book; to technical development editors Donald Bailey, Joel Kotarski, Jeanne Boyarsky, and John Hyaduck; and to Romin Irani, technical proofreader.

This book was significantly improved by my editor at Manning, Dan Maharry, who helped to develop and edit the book from concept to finished product. I'd like to thank Dan for his excellent insights, support, encouragement, and guidance throughout the process.

A big thank you to all the technical reviewers who read the manuscript at various stages of its development and contributed invaluable feedback: Adam Hinden, Antonio Mas Rodriguez, Betsy Hoofnagle, Catherine Freytag, Dr. Christian Mennerich, Dan Kacenjar, David Kerns, Ema Battista, Fanick Atchia, Grace Kacenjar, Henry Freytag, Jaqueline Currie, John Pentakalos, Keenan Hom, Kevin Adjaho Atchia, Matthew

Giblin, Nathan Sperry, Odysseas Pentakalos, Sam Kerns, Richard Freytag, Savannah Wilson, and Scott M. King.

Thank you also to all the readers who bought and read the MEAP (Manning Early Access Program) versions of the chapters and who took the time to post comments in the Author Online forum. You helped make this a better book!

The Raspberry Pi Foundation, original inventors, and community deserve a special mention. Thank you for designing something that is helping children to learn computer science. I'd also like to thank Guido van Rossum, the inventor of Python; the Python Software Foundation; and the Python user community, for creating and maintaining a simple and useful programming language for everyone.

About this book

The Raspberry Pi is a small, low-cost computer invented in the U.K. by the Raspberry Pi Foundation. It provides an easy-to-use tool for learning to program in Python. The Raspberry Pi, with its companion memory card, is preloaded with all the software you need to jump into programming in Python. The Raspberry Pi is made for you to learn to code by playing with it. It includes many input and output ports to give you flexibility in how you connect it. Much like a desktop computer, you need to connect a keyboard, mouse, monitor, and power cable to get started.

This book will teach you how to set up your Raspberry Pi, to write programs in Python, and to use your Raspberry Pi and Python to complete some projects. We'll cover the basics of Python: displaying text, gathering input, repeating commands, creating logic, as well as using the input and output pins of your Raspberry Pi for projects.

This book does not cover advanced Python topics, nor act as a comprehensive reference for Python. Since it is a book for beginners, these topics have been left out for clarity and brevity. If you'd like to learn more Python, there are links to online resources throughout the book.

This book is for kids and other beginners who would like to learn to program. It's also for kids who have a Raspberry Pi and want to learn what they can do with it. We'll introduce you to your Raspberry Pi and teach you Python in a natural, playful way, introducing topics and giving you activities to do using your Raspberry Pi. You don't need to have any prior programming experience. As long as you know how to use a mouse and open up programs by clicking on icons or menu items, you'll do great.

This book requires a Raspberry Pi, cables, and some other parts to complete the projects and activities. These items are needed throughout the book:

- Raspberry Pi 2 Model B
- 8 GB SD memory card, preloaded with the Raspberry Pi Foundation's NOOBS (New Out of the Box Software)
- USB power supply with micro USB cable (must deliver 1.2 A @ 5 V)
- USB keyboard
- USB mouse
- TV or monitor
- Cable to connect to TV or monitor (specific cables for your TV or
- monitor are discussed in chapter 1)

To complete the projects in part 3, you'll also need these parts:

- Solderless breadboard
- GPIO ribbon cable for the Raspberry Pi 2 Model B (40 pin)
- GPIO breakout board
- 1 dozen jumper wires, male-to-male
- 1 red LED (light-emitting diode)
- 1 green LED
- 1 blue LED
- 1 red, green, blue (RGB) LED
- 3 push buttons
- 3 resistors, 10K ohm
- 3 resistors, 180 ohm (or between 100 and 300 ohms)
- Headphones or powered computer speakers

You can typically find all these items in a Raspberry Pi starter kit or available individually through online retailers and stores that sell the Raspberry Pi, such as CanaKit, Sparkfun, or Adafruit.

Roadmap

This book is divided into three parts.

Part 1 introduces you to the Rasperry Pi, shows you how to set it up, and provides an introduction to the Python programming language:

- Chapter 1 provides an overview of the Raspberry Pi and how to set it up for the first time.
- Chapter 2 shows you how to write your first Python programs and introduces you to doing math and displaying text with Python.

Part 2 shows you how to build different text-based games while learning how to gather input, display information, make decisions, and repeat instructions in Python:

- Chapter 3 teaches you how to create your first interactive Python game, the Silly Sentence Generator 3000, by asking users to type in something and then displaying funny messages to the screen.
- Chapter 4 explores how to give your programs logic and use repeating loops as you create a Norwegian Blue Guessing Game.
- Chapter 5 demonstrates how to build a Cave Adventure Game, give users multiple choices, check input from users, and create your own Python functions.

Part 3 involves making your Raspberry Pi interact with the world around it:

- Chapter 6 explains setting up your Pi with an electronics breadboard, building a simple circuit, and controlling an LED (light) using your Raspberry Pi and Python.
- Chapter 7 dives into creating an interactive guessing game that uses lights to respond to a player's input, letting them know with different colors whether their answer is right or wrong.
- Chapter 8 teaches you how to listen to your Pi's input pins by making a project that combines light and sound to make your own DJ Raspi sound mixer.

Code conventions and downloads

All source code in this book is in a `fixed-width font like this`, which sets it apart from the surrounding text. In many listings, the code is annotated to point out key concepts. I have tried to format the code so

that it fits within the available page space in the book by adding line breaks and using indentation carefully.

The code accompanying this book is hosted at the GitHub repository: https://github.com/rheitz/hello-raspberry-pi. It is also available for download as a zip file from the publisher's website at www.manning.com/books/hello-raspberry-pi.

Author Online

Purchase of *Hello Raspberry Pi!* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and other users. To access the forum and subscribe to it, point your web browser to www.manning.com/books/hello-raspberry-pi. This Author Online (AO) page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog among individual readers and between readers and the author can take place. It's not a commitment to any specific amount of participation on the part of the author, whose contribution to the AO remains voluntary (and unpaid). We suggest you try asking the author some challenging questions, lest his interest stray!

The AO forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the author

Ryan Heitz is a teacher, programmer, maker, father, and big kid. He is the cofounder of Ideaventions, a Science Center for kids, and Ideaventions Academy for Mathematics and Science, a private school focused on science and technology. He specializes in teaching kids how to experience computer science in a fun and engaging way. As a programmer, Ryan has developed software for everything from NASA data collection systems to web mapping applications.

Part 1

Getting started

Get ready to explore Python using your Raspberry Pi! You'll need a Raspberry Pi and a few other parts and cables for part 1. Here's your shopping list:

- Raspberry Pi 2 Model B
- 8 GB SD memory card, preloaded with the Raspberry Pi Foundation's NOOBS (New Out Of the Box Software)
- USB power supply with micro USB cable (must deliver 1.2 A @ 5 V)
- USB keyboard
- USB mouse
- TV or monitor
- Cable to connect to TV or monitor (specific cables for your TV or monitor are discussed in chapter 1)

Optional item:

- Raspberry Pi case

Part 1 will get you on your way to using your Raspberry Pi and launch you into programming it with Python. In chapter 1, you'll set up your Raspberry Pi, learn how to start (or boot) it up, and then look around inside the Pi's desktop. Chapter 2 is where you'll start exploring the Python language. You'll create your first programs and learn to give instructions to your Raspberry Pi using Python.

By the end of part 1, you'll know how to get a Raspberry Pi up and running. You'll be able to write a Python program and interact with your Pi to make it do things like figure out the cost of a cheeseburger meal and display silly messages on the screen.

1

Meet Raspberry Pi

In this chapter, you'll learn how to

- *Set up your Raspberry Pi*
- *Install an operating system—Raspbian—on your Pi*
- *Find and open applications*
- *Write your first bit of code in Python*

What kinds of things do you think you can do with a Raspberry Pi?

1. Play games.
2. Watch videos.
3. Create a video game.
4. Listen to music.
5. Make a sound mixer for a dance party.
6. Build a robot.

Believe it or not, these are all projects you can do yourself, and if you learn to program in Python, the sky is the limit. You can achieve quite a lot on your Pi, as long as you can write a program to do it. But before we talk about that, let's take a look at a Raspberry Pi and discover what makes it tick.

What is the Raspberry Pi?

The *Raspberry Pi*, sometimes referred to as the *Pi*, is a small, low-cost computer invented in the U.K. by the Raspberry Pi Foundation. It provides an easy-to-use tool to help you learn to code in Python (the *Pi* part of its name came from the focus on using it to code in Python).

About the size of a deck of cards, it isn't as powerful as a laptop or desktop computer; its computing power is more similar to that of a smart phone. But what it lacks in processing power, it makes up for in its many features:

- Its readiness for programming in Python
- The many ways you can use it
- Its small size and cost

The Pi, with its companion memory card, is preloaded with all the software you need to jump into programming in Python. Type in commands, and see what happens. Enter a program you find on the internet or in a magazine, run it, and see how it works. The Pi is made for you to learn to code by playing with it, using it, and interacting with it.

Once you learn to program in Python, you can use your Pi as a base for all sorts of projects—with your imagination, the possibilities are endless! The Pi's small size makes it easy to carry around and include in projects. Hide it on a shelf or mount it on a wall with a camera to make a security system; power it with a rechargeable battery pack if you need it to be portable; or even attach it to a remote-controlled car or helicopter. And if you happen to mess something up, it's simple to recover. Even if you manage to break the Pi, it's pretty cheap to replace.

At its core, the Raspberry Pi is a circuit board that has all the components found in many computers. The next section checks out the components of the Pi and explores what they do. Let's go!

Exploring your Raspberry Pi's parts: hardware

Ever look closely at an insect under a magnifying glass, or take apart a toy? Humans are naturally curious about what makes things work. What are the different parts, and what do they do? What parts are

unique? Let's treat the Raspberry Pi the same way, explore its parts, and learn how to set it up.

Luckily, you don't have to break it open to see its parts. You can see the Raspberry Pi's components displayed before you on the green circuit board in your hand (see figure 1.1). Let's walk through the parts of the Raspberry Pi and see what they do. We'll be focusing on the Raspberry Pi 2 Model B; if you have a Raspberry Pi 1 Model B+ or B, see appendix B for more information.

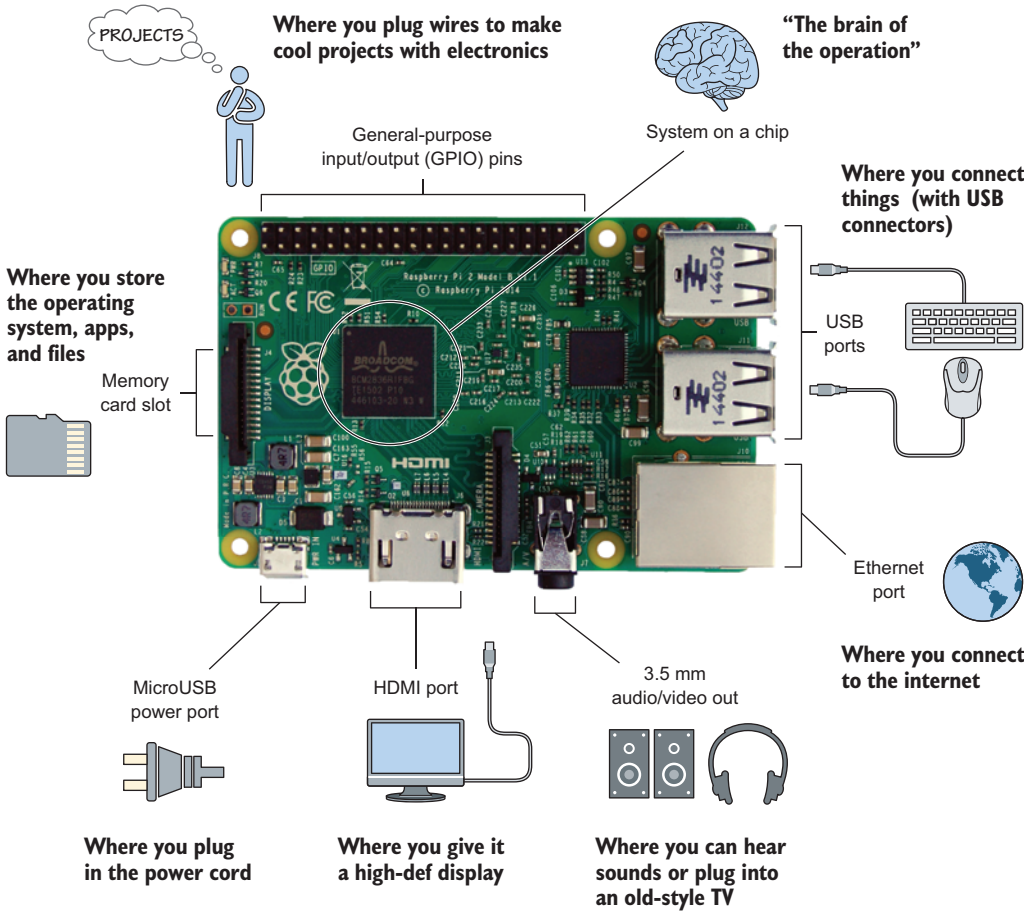


Figure 1.1 The Raspberry Pi provides an excellent platform for learning to program in Python. It includes many input and output ports to give you flexibility in how you connect it. As you would with a desktop computer, you need to connect a keyboard, mouse, monitor, and power cable before you can start using your Pi.

Defining some tech terms

Input and *output* are terms used for communication to and from a computer.

USB refers to a common connector found on computers. It's used to plug in a keyboard, a mouse, flash drives, and many other computer peripherals.

HDMI is a standard way to connect devices to high-definition TVs or monitors. We'll talk about this more later, when we discuss connecting a TV or monitor to your Raspberry Pi.

Ethernet is a technology used to connect computers together into a network. This port provides a way to plug in and connect to the internet or your home network if a wireless connection isn't available.

Giving your Pi a cozy home: Pi cases

We all like to be warm and cozy in our homes. A Raspberry Pi is no different. Do the right thing and protect your Pi by putting it in a case (see figure 1.2). If your Pi didn't come with a case, you have a lot of options. You can buy one or make your own. My favorite approach is to make my own case from wood, cardboard, a plastic container, or even LEGOs. The key is making sure your Pi is protected from accidental drops and, ideally, spills. But before you close up your Pi in a case, let's take a closer look at some of its features.

Figure 1.2 A case protects your Raspberry Pi from damage while making it easy to access the ports. Some people use a case to give their Pi a unique personality. You can purchase a case or, better yet, make your own. Plastic cases are the most common, but these pictures show examples of cases made from paper, plastic, and aluminum. You could even try using LEGOs to make one.

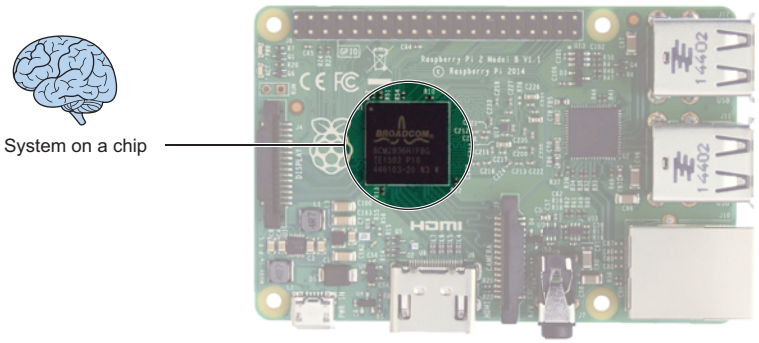


Figure 1.3 The Raspberry Pi's system on a chip (SoC) contains its computing and graphics processing power and working memory. The Pi uses the ARM11 microprocessor as its CPU and the VideoCore IV for its GPU. The ARM11 microprocessor is found in handheld electronics such as smart phones and gaming systems. The SoC in the Raspberry Pi 2 Model B comes with 1 GB of RAM.

The brain of your Pi: system on a chip

Meet the brain of your Raspberry Pi. The *system on a chip (SoC)* is the black square in the middle of the Pi circuit board in figure 1.3. This incredible chip is a package of many parts: the central processing unit (CPU), the graphics processing unit (GPU), the digital signal processor, and the Pi's working memory. The chip provides the computing power, graphics power, and memory to run apps and play videos.

The Pi's CPU handles running applications and executing instructions. The same processor is also found in smart phones and e-readers. Think of it as the part of your brain that allows you to follow instructions and calculate the answer to math problems.

The GPU is like the visual part of your brain that allows you to visualize a 3D object in your mind or track a ball thrown to you. It handles the Pi's multimedia tasks, like processing digital images, drawing graphics, and playing videos. The GPU gives your Pi surprisingly good high-definition video-playback capabilities. Both the central processor and the graphics processor share the Pi's working memory, or RAM, which is part of the SoC.

Working memory: RAM

Question: Can you remember the following grocery store list? *Bananas, milk, peanut butter, jam, bread.* Read the list once more, and then look away from the book and try to recite the list from memory.

To remember it, you need to hold the names of the items in your memory. You only have to store them for a short time. Once you go to the store and buy the items, you can forget them.

When a computer is working, it does much the same thing. It may have to remember and process millions of instructions and bits of information each second, but it can often forget them once it's done processing them. The computer does this using working memory or *random access memory (RAM)*. It's packed in the SoC, and it gives your Raspberry Pi the ability to process instructions quickly by remembering pieces of information as it's working and forgetting them when they're no longer needed—much like how the neurons in your brain work together to remember a grocery list. Later, we'll talk about storing information for the long term and where that happens.

Connecting a keyboard and mouse: USB ports

Meet the *USB* ports on your Raspberry Pi. The two metal, rectangular boxes each contain two USB ports, shown in figure 1.4. USB stands for *Universal Serial Bus*.[1] The Pi provides USB ports to allow you to connect a keyboard, a mouse, flash drives, and other USB peripherals.

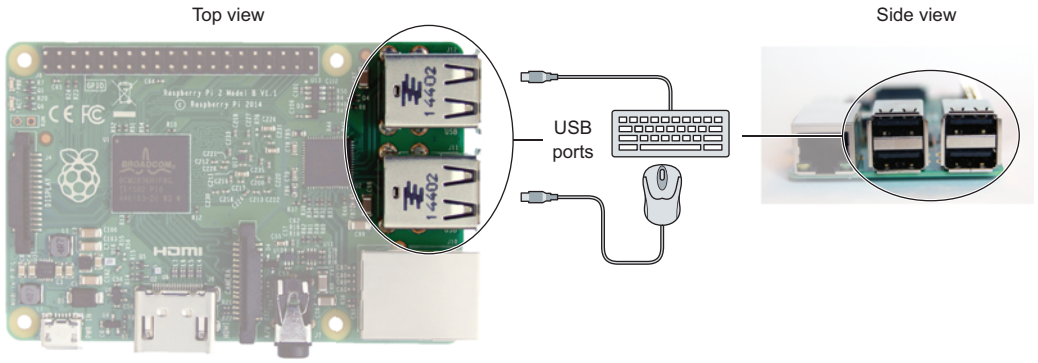


Figure 1.4 The Raspberry Pi 2 Model B has four USB ports. They're on the board in two sets of two, side by side. The USB ports are useful for connecting a keyboard and mouse to your Pi. A USB hub can also be plugged in to allow for even more peripherals.

[1] The *U* for *Universal* is because it provides computer makers and computer equipment makers with a standard way to connect things to computers. Things connected to a computer are often called *peripherals*.

Why are they called ports?

Back in ancient times, when Romans walked around and spoke Latin to each other, the word for a gate or door was *porta*. Although computers don't have doors or gates, they have places where you plug things in, called *ports*.

Ports allow electrical signals to go in and out of your computer. Without ports, you wouldn't be able to view your computer's screen, download web pages, or move a mouse.

Let's pretend you could shrink and that you had special glasses so you could see these electrical signals. What would you see when I pressed the E key on the keyboard? You'd see an electrical signal flying from the keyboard through the keyboard's wire, through the port on the computer, and into the computer. The port acts like a gate, allowing signals to go into or out of your computer.

Get your keyboard and mouse. Let's plug them into your Pi.

CONNECTING A KEYBOARD

You'll need a keyboard that plugs into a USB port. Figure 1.5 shows an example of a keyboard with a USB connector.[2]

To attach your keyboard to your Pi, plug the wire from your keyboard into your Raspberry Pi's USB port. There are four USB ports on your Pi. It doesn't matter which one you choose.

Figure 1.5 You need a USB keyboard to type and enter commands on your Raspberry Pi. The keyboard plugs into one of the four available USB ports on the Raspberry Pi 2 Model B.

[2] If you don't have a keyboard with a USB connector, have no fear. You can find one for under $15 online or at your local computer or electronics store.

> TIP If the keyboard's USB connector doesn't fit into the Raspberry Pi's USB connector, flip over the connector and try again. USB connectors only fit in one way.

Fantastic! Your keyboard is connected to your Pi. It's time to move on to adding a mouse.

CONNECTING A MOUSE

For this step, you need a mouse that plugs into a USB port. The keyboard is using one of your Raspberry Pi's four USB ports. Plug your mouse into one of the other ports.

ANOTHER OPTION: WIRELESS KEYBOARD AND MOUSE COMBINATION

If you own a wireless keyboard and mouse combination, instead of using wires, you can plug the USB dongle into one of the USB ports on the Pi. This frees up one of your USB ports, which can be handy should you decide to attach multiple USB devices such as a USB Wi-Fi adapter or USB flash drives, or if you want fewer wires on your desk.

Excellent! Giving your Pi the ability to store and retrieve information is your next task.

Storing memories: your Pi gets a memory card

We all like to remember things that are important to us. Birthdays, vacations, and holidays are wonderful times, and we've invented ways to help us recall them. You might use a scrapbook or a photo album to store memories. Even after many years, you can open these books and remember these past events.

In addition to working memory (RAM), computers also need a way to remember things, even if they're turned off for long periods of time. The Raspberry Pi, like all computers, has this capability for memory storage, letting it save and retrieve data, files, and applications. Much like a photo album lets you recall holidays, the Pi's memory storage allows you to store important applications and information. You'll use this capability when you learn how to save sets of Python instructions or programs.

SD MEMORY CARD

A Raspberry Pi is different from most computers because its memory storage is contained on an SD memory card, whereas most laptops and desktops use a hard drive. Files, applications, and even the Pi's operating system are all stored on the SD memory card, whether it's a Python game you're creating or a new music player app for your Pi. If you purchase a Raspberry Pi kit, it will come with an SD card (see figure 1.6).[3]

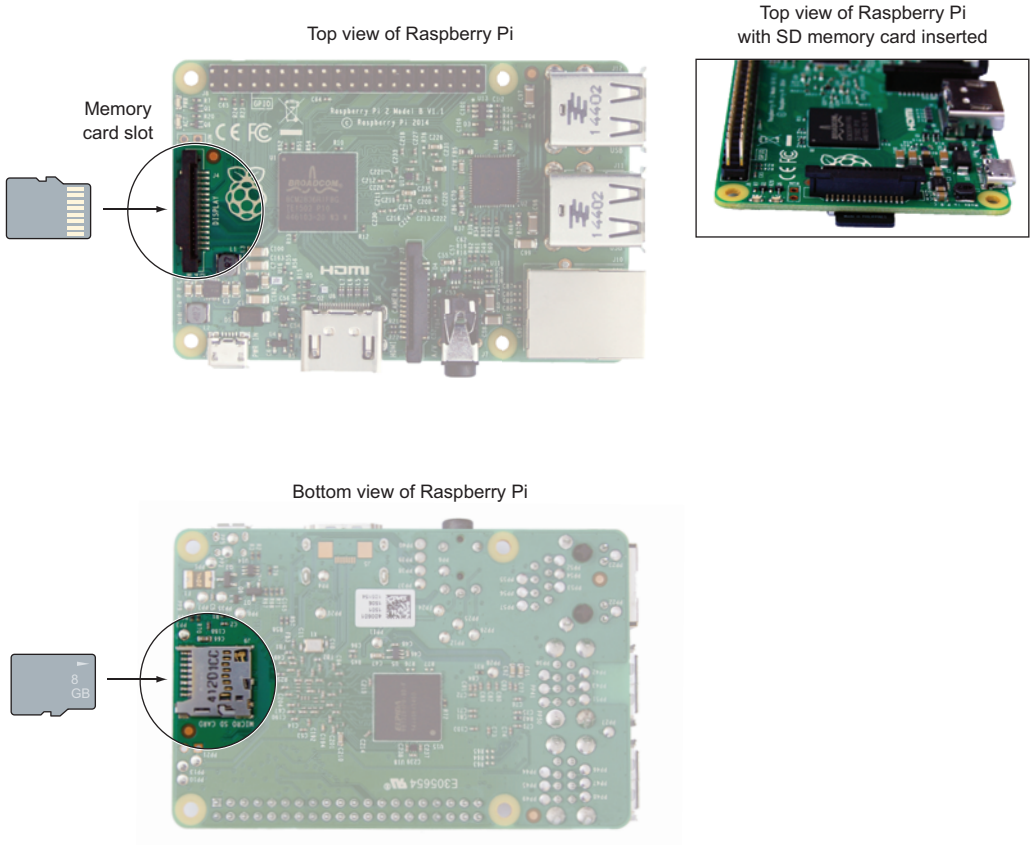


Figure 1.6 An SD memory card provides the storage memory used by the Raspberry Pi to hold all the software and files, including the operating system. Raspberry Pi kits come with an SD memory card preloaded with the software needed to start up your Pi. The two left images show the location of the SD memory card slot on the underside of the Pi board. The right image shows an SD memory card inserted into the SD card slot.

[3] See http://elinux.org/RPi_SD_cards for more information on compatible cards.

SD cards come in various sizes

SD cards come in three sizes: the full-size SD card (largest), the miniSD, and the microSD (smallest). The Raspberry Pi 2 Model B uses a microSD card.

You can add more storage to your Pi by attaching USB peripherals such as a USB flash drive or a USB hard drive.

NOOBS

Your Raspberry Pi kit comes with an SD card preloaded with NOOBS. Developed by the Raspberry Pi Foundation, *New Out of the Box Software (NOOBS)* is a set of files that helps you set up your Pi for the first time. If you lose yours or need a NOOBS SD memory card, you can buy new ones online. Alternatively, if you have an SD card and want to install NOOBS on it, go to the Raspberry Pi Foundation website (www.raspberrypi.org/downloads) to learn how.

SD MEMORY CARD SLOT

Figure 1.6 shows the location of the SD memory card slot. This thin, metal slot is on the underside of the Raspberry Pi. For your Pi to work when you plug it in, it must have some initial knowledge to start up and display something on the screen. In addition to this startup information, it must also have a place to store any new information.

INSERTING THE SD CARD IN THE SLOT

Hold the card so that the end with the metal contacts is facing up and toward the Pi. Insert the card along the underside of the board into the slot. You'll hear a small click as the card is pushed into the slot. The card is held in place by a small spring mechanism. The card will only fit in one way, so if it doesn't fit, flip it over. If you need to remove the card, push it in again (you'll hear a click); then you can pull it out.

REPLACING A LOST OR BROKEN SD CARD

If you lose your SD card, you lose the information, applications, and operating system that are stored on the card. It's as if you lost your hard drive on a home computer. You can easily replace the card, but

you'll be starting over fresh. Here are the two options for replacing the card:

- Purchase an SD card at the store, and set it up anew. It's recommended that you get an SD memory card with at least 8 GB of storage space. You can download and install the startup software from the Raspberry Pi Foundation at www.raspberrypi.org/downloads. See appendix A for instructions on how to make a new SD card for your Raspberry Pi.
- Buy an SD memory card preinstalled with the Raspberry Pi startup software. You can find cards for sale on the Raspberry Pi Foundation website and at online retailers.

SD CARDS MAKE YOUR PI'S MEMORY PORTABLE

If your Raspberry Pi ever breaks, you can remove the SD memory card and insert it into a new Pi. All your files and software will be there. It's like taking your photo album with you to a new house. The memories are safe in the photo album, ready for you to enjoy.

> TIP You can set up multiple SD cards for your Raspberry Pi and switch them whenever you want to give your Pi a whole different personality. Maybe set up an SD card for the Pi as a media center, complete with games, music, and videos. Set up another for your Pi robot project. Each memory card can be set up uniquely, with different operating systems, applications, and files. Swap out the SD card and reboot your Pi, and you instantly have a Pi with different traits to meet your needs.

Connecting a TV or monitor: HDMI port

The *HDMI* port, shown in figure 1.7, is for connecting your Raspberry Pi to a TV or monitor. HDMI stands for *high-definition multimedia interface*. The output provides a combined audio and video signal—meaning both sound and picture come out of this port and go to your TV or monitor. If you want a crisp, clear display and you already own a high-definition TV or monitor, then you'll want to connect your Raspberry Pi to it using the HDMI output port. Because the HDMI output contains audio and video signals, if your TV or monitor has built-in

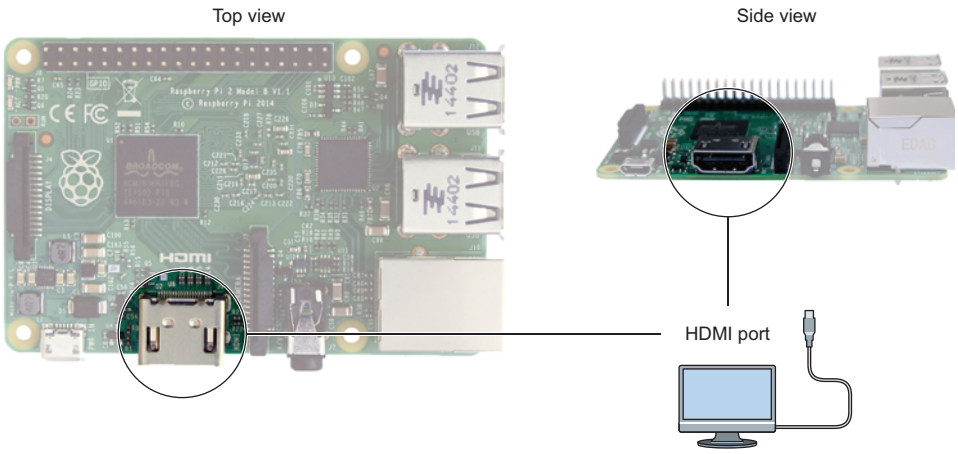


Figure 1.7 The HDMI port on the Raspberry Pi provides a high-definition audio and video signal that can be connected to a TV or monitor. Use an HDMI cable to connect your Pi to your TV or monitor. Depending on the connectors available on the TV or monitor, you may need an adapter.

speakers, the sound from your Raspberry Pi can be set to come out of the speakers rather than through the 3.5 mm audio output.

Now that you know about the HDMI port, let's see how you can connect your Pi to a TV or monitor.

CONNECTING YOUR PI TO A TV OR MONITOR

Once you decide on the TV or monitor you plan to use, you'll need to look for the available video input ports on the TV or monitor (look on the back or sides to find them). What kinds of ports do you see? Unfortunately, manufacturers often provide a variety of different ports. Think of it like a matching game. Your goal is to match the connectors on your TV to the connectors on the Pi. If they don't match, you'll need to use one of the adapters discussed in a minute. Either way, you're sure to get it solved.

IDENTIFYING PORTS AND MAKING THE CONNECTION

Take time to study the connections on your TV or monitor. Try to identify the video ports, comparing them to the pictures of connectors in figure 1.8.

This section provides instructions on how you can connect your Pi to a TV or monitor with either an HDMI or a DVI port. If your TV or monitor has different video input ports, check appendix B for tips on connecting to them.

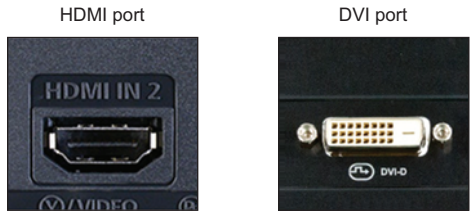


Figure 1.8 HDMI and DVI are common types of video input ports found on modern TVs and monitors. It's easiest to connect a Raspberry Pi to a TV or monitor with an HDMI port. HDMI provides a high-definition picture and doesn't require any adapters or converters—only an HDMI cable, which is included in many Pi kits. The DVI port requires a special adapter to connect with a Pi.

HDMI

The HDMI port is a metal, mostly rectangular port that is labeled *HDMI*. Connect an HDMI cable from the screen's HDMI port to your Raspberry Pi's HDMI port (see figure 1.9). If you've connected your HDMI cable, you can now skip ahead to the discussion of other ports on the Pi.

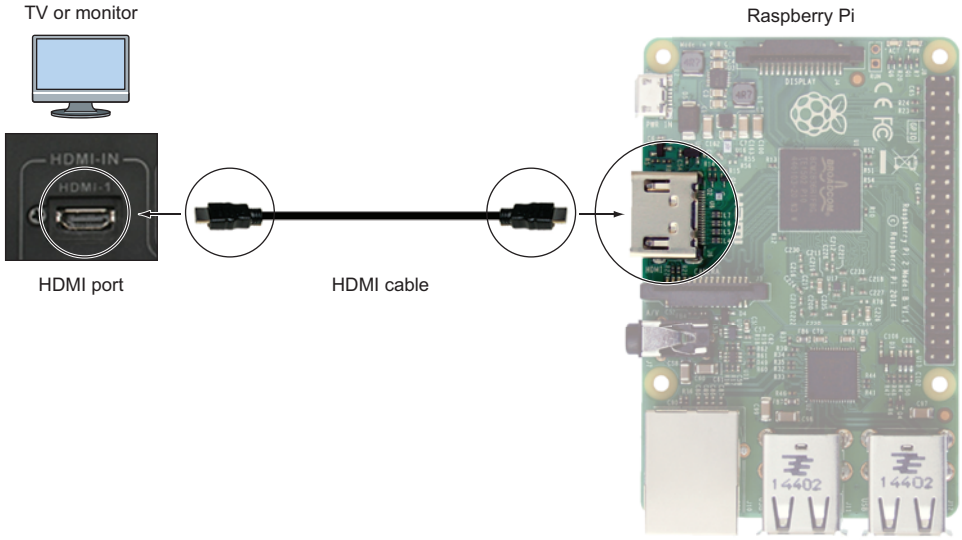


Figure 1.9 A Raspberry Pi can be connected to a TV or monitor using an HDMI cable. Connect the cable from the Pi's HDMI port to the TV's or monitor's HDMI input. In addition to video, the HDMI cable also contains the Pi's audio output, which can be played through the TV's or monitor's speakers.

DVI

DVI ports on TVs and monitors come in several different forms. They're all rectangular ports with three rows of eight square pinholes and a horizontal hole or set of holes next to them. If you already have an HDMI cable, the solution is to purchase an HDMI-to-DVI adapter. You can find these online or in a computer store. Plug the adapter into the computer screen's DVI port, and then plug your HDMI cable into the back of the adapter and the other end into the HDMI port on your Raspberry Pi (see figure 1.10).

Another solution, rather than to use an adapter, is to purchase a DVI-to-HDMI cable. These can be found online or at a computer store. Plug the DVI connector on the cable into your computer screen, and plug the HDMI connector into your Pi's HDMI port.

Great! You've completed an important step by connecting your Pi to a TV or monitor.

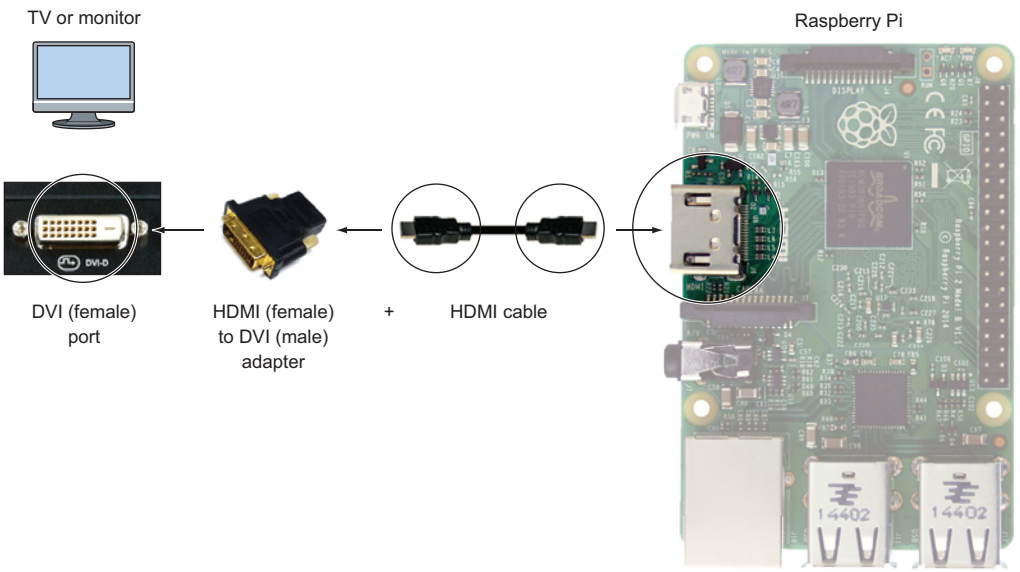


Figure 1.10 The Raspberry Pi can be connected to a TV or monitor with a DVI port using an HDMI-to-DVI adapter and an HDMI cable. One end of the HDMI cable plugs into the Pi's HDMI port. The other is connected to the adapter, and the adapter is connected to the TV or monitor. Adapters are available through online retailers or local computer stores.

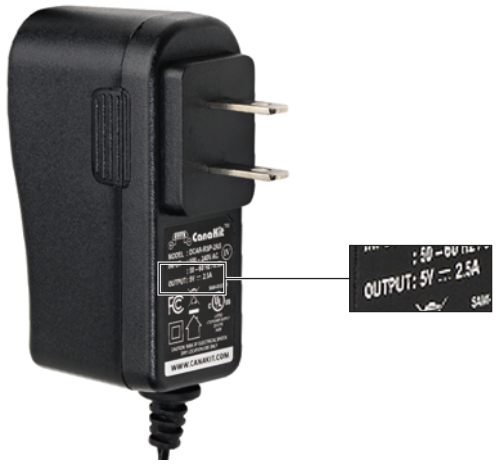
Other ports and connections

You'll find other ports on your Raspberry Pi. We'll cover those in later chapters, or you can reference appendix B for more information on specific ports and connections. Some of these include the following:

- *GPIO pins*—The two long rows of pins on the Raspberry Pi are used to send and receive electrical signals. Part 3 of this book will cover how to program those pins and build projects.
- *Internet*—You can connect your Raspberry Pi to the internet or your home network by plugging in an Ethernet cable. But you may find that the easiest way to get online is to use the USB Wi-Fi adapter that is provided in many Raspberry Pi kits. Appendix B has information on the Ethernet port and using USB Wi-Fi adapters.
- *3.5 mm audio/video out*—The small round connector is for plugging in headphones or powered speakers. Chapter 8 will show you how to play sounds as you turn your Raspberry Pi into a music player.

Let's see how you can get power to your Pi.

Powering your Pi: microUSB power port



Power for your Raspberry Pi is supplied through the microUSB power port located near a corner of the board (see figure 1.11). This port is where you connect a power supply to your Pi; it's the same as the port found on many mobile phones. Raspberry Pi kits come with a microUSB power supply.

Figure 1.11 The Raspberry Pi requires a microUSB power supply that provides at least 1.2 A of electric current. If you plan to use all the USB ports on your Pi, you may want one that provides 2 A or more of electric current. The recommended voltage is 5 volts (V), but the Pi can operate at voltages ranging from 4.8 to 5.2 V. If you have a power supply you want to use with your Pi, check its output voltage and current, which are listed on the charger in small print. In this example, the charger has an output of 5.1 V and 2.5 A of current, making it a suitable power supply for a Pi. Using the incorrect voltage or insufficient current can damage or destroy your Pi, so check carefully.

NOTE Only certain mobile phone chargers can be used to power your Raspberry Pi. The charger must produce sufficient electrical current to power it. If you want to go this route, then you should read the fine print on the charger. The charger must produce 1.2 amp (A) or more for the Pi.

It's alive! Plugging in the Pi

Before plugging your Raspberry Pi into the power supply, go through this quick checklist:

1. Are you sure your keyboard, mouse, and monitor are connected to the Pi?
2. Have you turned on your TV or monitor and set it to the correct input source? For example, if you plugged your Raspberry Pi into the TV's HDMI port, make sure the TV is set to HDMI input.
3. Have you inserted your SD card with NOOBS into your Pi?

An example setup is shown in figure 1.12.

Figure 1.12 Example setup of a Raspberry Pi with peripherals connected and SD card inserted. A keyboard and mouse are connected to the Pi's two available USB ports. A microUSB power supply is plugged into the Pi; the other end is lying on the desk, ready to be plugged into the wall. An HDMI cable is connected from the Pi's HDMI port to the back of the monitor. The Ethernet port has an Ethernet cable plugged into it from a router (not shown).

TIP TVs and monitors often allow you to connect multiple video sources. Maybe your TV has a Wii, a DVD player, and a digital video recorder. These TVs and monitors have the option to select which input is displayed to the screen. Use your TV's or monitor's input selector to set the correct input.

All right, if you have all three steps checked off, it's time to power up your Raspberry Pi. Plug your power supply into a wall outlet, and plug the microUSB connector into your Pi. Your Pi's lights will begin to flash. Enjoy the beautiful glow from the lights—this is a sign that your Raspberry Pi is starting up. It's also referred to as *booting*; this is when the computer detects the devices you have connected to it and starts up the computer's operating system (OS). Some believe the term *boot* originated from kicking a horse to get it to start moving. You can imagine that you're giving your Pi a bit of a boot to get it started.

Getting your Pi running: software

You've got your Pi plugged in and ready to rock. It's time to get it running and doing something useful—and for that, you need some software.

An OS is a common set of instructions, or software, that helps manage the computer. Common OSs you've most likely encountered are Microsoft Windows, Apple's OS X, and Linux. All of these OSs control the connection of your keyboard, mouse, monitor, and other peripherals. Most important, the OS serves as a foundation for you to put applications on your computer and use them.

The SD memory card that comes with your Pi kit already contains the files for installing several different OSs on your Pi. We'll step through installing the Raspbian OS—the default for the Pi—and configuring it.

Installing the Raspbian operating system

The first time you boot a Raspberry Pi, you'll need to install an OS on it and then configure it to work nicely for you. Let's walk through the first task: installing an OS. You'll configure it in the next section. Once you plug in your Pi, you'll see the NOOBS menu for selecting an OS, as shown in figure 1.13.

The Raspberry Pi has a variety of OSs that can be installed on it. The Raspberry Pi Foundation recommends the Raspbian OS, and it's what we'll use for this book. Let's go over how to install it on your Pi.

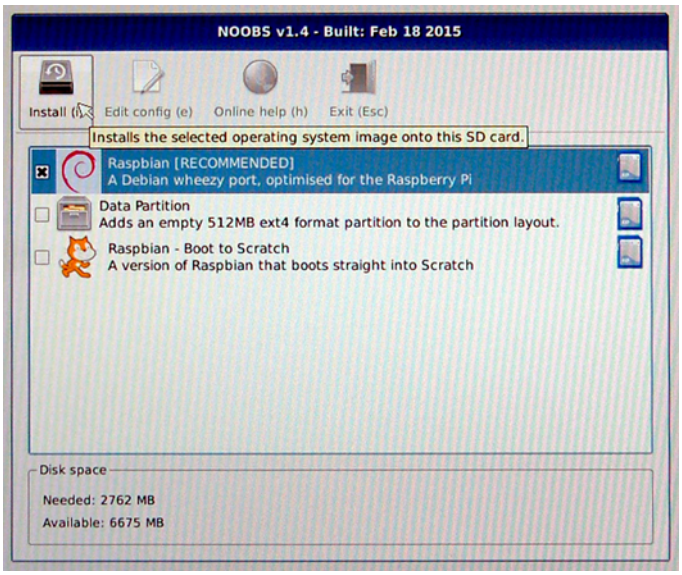


Figure 1.13 The NOOBS selection menu allows you to choose the OS you want to install on your SD card and use with your Raspberry Pi. This menu appears the first time you start up your Pi.

What if you don't see the NOOBS software screen?

If you don't see the NOOBS software screen after your Pi boots up for the first time, then there are a few things to check.

If you don't see lights flashing on your Pi when you plug it in, make sure the electrical outlet you're using has power. Many a Pi owner has accidentally plugged a Pi into a power strip and forgotten to switch on the power strip. Sounds silly, but even the best programmers make mistakes.

If your Pi's lights blink when you plug in the power supply but the screen of your monitor doesn't show anything, make sure the monitor is plugged into an electrical outlet, the HDMI cable is connected from the monitor to the Pi, and you've turned on the monitor.

Finally, if your Pi starts booting up and you see lots of messages displaying on a black screen, but you never see the NOOBS selection menu, it's likely that your SD card has an error. See appendix A for ways to fix an SD card.

Sometimes you'll run into issues with your Pi. If you do, use the troubleshooting steps in appendix A, and search the Raspberry Pi Foundation website[a] to find solutions.

[a] The Raspberry Pi Foundation website is www.raspberrypi.org.

On the NOOBS selection menu (see figure 1.13), follow these steps:

1. Select Raspbian (make sure there is an X in the box next to Raspbian; if not, click the box to select it).
2. Click the Install button at the top of the menu.
3. A message appears, warning you that the process will install the OS and that all existing data on your SD card will be overwritten.[4] Select Yes to continue with the installation.
4. Wait for the installation to complete. It will take 5 to 10 minutes, so get a drink or grab a snack while you're waiting.
5. When the installation is done, a box pops up, letting you know the OS was installed successfully. Click OK, and your Raspberry Pi will start loading Raspbian.
6. When it's finished loading Raspbian, your Raspberry Pi reboots itself. A black screen appears, followed by many, many, many messages. Don't worry; the messages are the Pi performing its startup tasks, such as detecting the keyboard, mouse, and TV or monitor.

Kudos to you! You've installed your Raspberry Pi's OS, Raspbian. Now you'll want to configure how it works to suit you.

Configuring the operating system: making it yours

You've finished installing the Raspbian OS on your SD memory card and gotten it running for the first time. The next thing you'll see is the Raspberry Pi configuration screen, shown in figure 1.14.

> TIP You can't use your mouse with this menu! Use the arrow keys (up, down, left, and right) and Tab key to move around the menu instead. Press Enter to select the highlighted menu item.
>
> Let's walk through some of the basic configuration settings you may want to change.

[4] When you're warned that all data will be overwritten, this doesn't include NOOBS, which is retained on the SD card so that you can reinstall the OS if you ever need to.

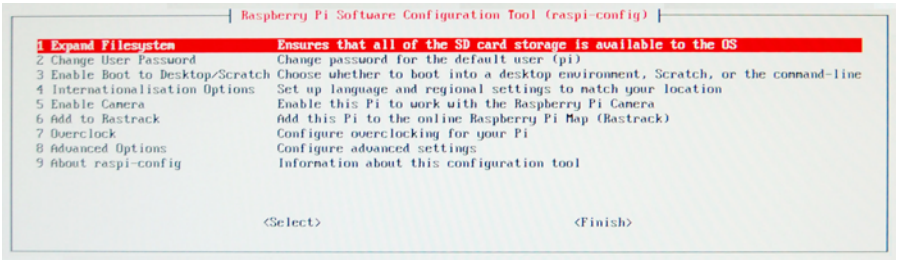


Figure 1.14 When your Pi boots up for the first time, you'll see the Raspberry Pi configuration menu. This menu makes it easier to set up your Pi by allowing you to change settings such as the time zone and keyboard layout. The menu also has the option to set your Pi to always boot to the Raspbian desktop environment.

CHANGING THE KEYBOARD SETTINGS

The Raspberry Pi is made in the U.K., so it's preset to a U.K. keyboard. If you live in other parts of the world, the keyboard may make unexpected characters appear on the screen. For example, you might type a # symbol (Shift-3), and your Pi displays the symbol for a British pound. Weird, right?

You can use the configuration tool to change your Pi's keyboard layout by following these steps:

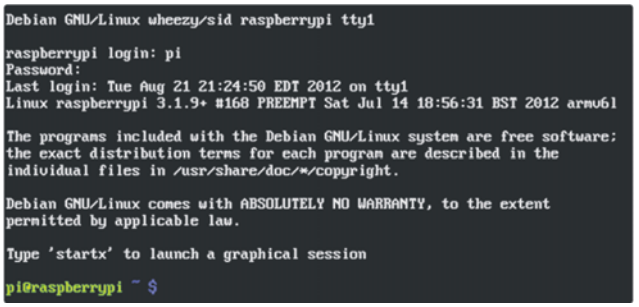
1. On the Raspberry Pi configuration menu, select option *4*—Internationalisation Options—and press Enter.
2. Select Change Keyboard Layout, and press Enter.
3. Select your keyboard model—for example, Dell—and press Enter.
4. You see options for the keyboard layout's country of origin. Select the appropriate country, and press Enter.
5. A list of keyboard layouts appears. Select the one for your location, and press Enter.
6. On the next series of screens, you can set shortcut keys. Set them to match your personal preferences. If you aren't sure, accept the defaults (press Enter until you're back to the configuration menu).

You can always return to the configuration tool if needed. You'll learn how in a later section when you're introduced to the command-line mode for Raspbian.

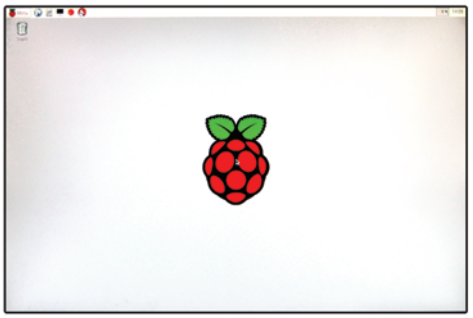
CHOOSING HOW YOUR RASPBERRY PI STARTS UP

Raspbian, like most OSs, allows you to use it in two different ways (see figure 1.15):

- *Command-line mode*—You type in commands to the OS. This can be tough for novices, because you need to know the commands and type them in exactly. Because this mode is more difficult to use, you'll only use it in this book when you need to run commands that require administrative or super-user permissions. For example, you'll need the command line when you make Python programs that use the GPIO pins or you want to alter your Pi's configuration.
- *Graphical-user-interface (GUI) mode*—Everything appears in windows, icons, and menus that are point and click. Just like on Windows and Mac computers, this will be your main way to interact with your Pi and program in Python. It represents the most natural way to access applications, files, and folders.



Raspbian command-line mode



Raspbian graphical-user-interface (GUI) mode

Figure 1.15 Example screen images of a command-line mode (top) and a GUI mode (bottom) for a Raspberry Pi running the Raspbian OS. The command-line mode is text-based: you enter instructions at the prompt. The GUI is pretty much the same as a Windows or Mac interface, with windows, icons, and menus that you interact with using a mouse pointer.

Question: Which option do you prefer?

- Your Raspberry Pi booting up to a screen with a blinking cursor, waiting for you to type in commands
- Your Raspberry Pi booting up and showing you a desktop with application icons arranged on the screen, waiting for you to point to and click them with your mouse

If you chose the second option, you can set Raspbian to always boot to the desktop with the following steps:

1. On the Raspberry Pi configuration menu, select option 3—Enable Boot to Desktop/Scratch—and press Enter.
2. Select the second option—"Desktop Log in as user 'pi' at the graphical desktop"—and press Enter.

Fantastic! Next time your Raspberry Pi boots up, you'll be taken to the Raspbian desktop.

> TIP If you decide you prefer to boot the Raspberry Pi to the command line, you can always launch the Raspbian desktop by entering `startx` at the command line.

> TIP Sometimes you may find yourself using the Raspbian GUI, but you want to use the command line. There is an easy way to change. You can open the command-line mode in a window by clicking the Menu Button, then selecting the Accessories category and clicking the Terminal[5] icon.

MAKING OTHER CHANGES

The Raspberry Pi configuration menu includes other options such as setting up a camera and over-clocking. These are available if you ever want to use them. Check the Raspberry Pi forums for more information on these options.

Saving your configuration and rebooting

If you're happy with the changes made to your Raspberry Pi, follow these steps to exit the Raspberry Pi configuration tool and reboot your Pi:

---

[5] Terminal is short for LXTerminal or Linux terminal. Raspbian is a Linux-based OS, and *terminal* refers to the command-line mode where you can enter commands.

1. On the Raspberry Pi configuration menu, use the arrow keys to select Finish, and press Enter.
2. You're prompted with this message: "Do you want to reboot now?" Select Yes, and press Enter.

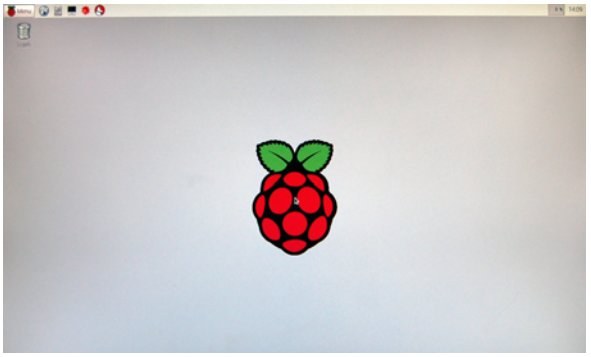


Figure 1.16 A view of the Raspbian desktop after your Raspberry Pi boots up. The desktop is similar to the desktop in Microsoft Windows or Apple Mac OS X. Don't worry if your desktop is different from this one. Depending on when you bought your Pi, you may have received an SD card with an older or newer version of Raspbian.

Your Raspberry Pi will display lots of lines of text as it boots up. (Yes, it does that again! Don't worry, it will seem normal to you soon.) This is your Pi's startup sequence when it connects peripherals and starts up the OS. Next, a white screen with a Raspberry Pi will appear, along with a set of icons—this is your Raspbian desktop (see figure 1.16). Congratulations! Your Raspberry Pi is ready to go.

A BIT OF PI IN YOUR FACE: TROUBLESHOOTING

If you don't see the view shown in figure 1.16, don't be discouraged. It's likely that you didn't select the option to boot to desktop. If your screen shows the command-line mode for Raspbian (figure 1.17), you can log in and launch the Raspbian GUI.

```
[ ok ] Setting up ALSA...done.
[info] Setting console screen modes.
[info] Skipping font and keymap setup (handled by console-setup).
[ ok ] Setting up console font and keymap...done.
[ ok ] Setting up X socket directories... /tmp/.X11-unix /tmp/.ICE-unix.
INIT: Entering runlevel: 2
[info] Using makefile-style concurrent boot in runlevel 2.
[ ok ] Network Interface Plugging Daemon...skip eth0...done.
[ ok ] Starting enhanced syslogd: rsyslogd.
[ ok ] Starting periodic command scheduler: cron.
[ ok ] Starting system message bus: dbus.
Starting dphys-swapfile swapfile setup ...
want /var/swap=100MByte, checking existing: keeping it
done.
[ ok ] Starting NTP server: ntpd.
[ ok ] Starting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 7 raspberrypi tty1

raspberrypi login:
```

Figure 1.17 If you didn't set up your Pi to boot to the Raspbian desktop, the command-line mode will be displayed when your Raspberry Pi boots up. It will ask you for your login name and password.

At the command line, you'll be prompted to enter your login and password. The default login is `pi`, and the password is `raspberry`. After entering that information, launch the Raspbian Desktop from the command line using the following steps:

1. Type `startx`.
2. Press Enter.

Once you execute the command, the Pi will start up the Raspbian GUI mode and display your Raspberry Pi's desktop. If you happen to have a different problem, head to appendix A for troubleshooting ideas.

Getting around: learning Raspbian

Take a cruise around your Raspberry Pi, and look at some of the applications that come already installed with the Raspbian OS.

Finding and opening applications on your Raspberry Pi

There are many applications on your Raspberry Pi. You can access them by clicking the Menu button in the top-left corner of the desktop (see figure 1.18). Enjoy exploring what comes installed on your Pi.

Your files and folders

Similar to Windows Explorer or Mac Finder, Raspbian has some built-in tools to make it easier to navigate the folders and files on your

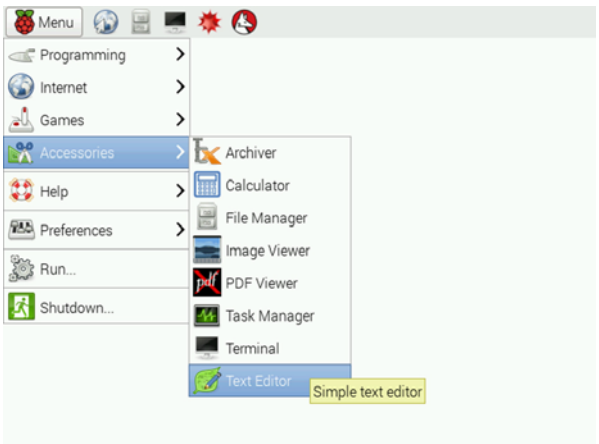


Figure 1.18 The Raspbian application menu opens when you click the Menu button in the top-left corner of the desktop. You can open an application by moving your mouse over the categories listed on the menu and then clicking the application.

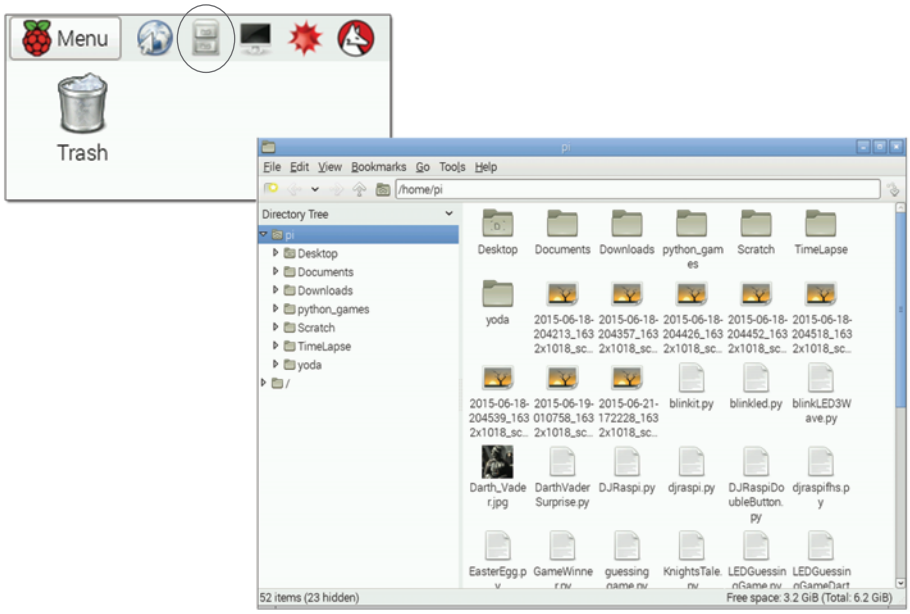


Figure 1.19 File Manager in Raspbian allows you to manage files as you do in Windows Explorer or Mac Finder. You access File Manager using the folder icon in the upper-left corner of the desktop. This is a view of a Pi with a lot of files stored in the /home/pi folder.

Raspberry Pi. In Raspbian, the application for managing files is called *File Manager*, and it's accessed by clicking the folder icon located in the top-left corner of the Raspbian desktop. Figure 1.19 shows the icon and the File Manager application. Just as in Windows Explorer, you can

- Navigate into folders by double-clicking them.
- Drag files to move them to another folder.
- Copy and paste files using the right-click menu on files and folders.
- Rename files.
- Open files by double-clicking them.

The Pi was built for coding. Let's see how you can write code on your Pi.

Writing code

You're going to learn to write code in the Python programming language. Meet a new program, IDLE. IDLE is a tool that'll help you write programs in Python. IDLE stands for Integrated DeveLopment Environment. The Python language was named after Monty Python, and the IDLE acronym is a nod to Eric Idle, one of the founding Monty Python members.

Follow these steps:

Click the Menu button on your desktop.

Select Programming > Python 3.

After a second or two, IDLE opens the Python Shell, as shown in figure 1.20.

NOTE Previous Raspberry Pi models have desktop icons for Python: IDLE and IDLE 3. *You'll use Python 3 (or IDLE 3) for the exercises in this book.* On older Pi models, the IDLE 3 icon opens the Python Shell for Python 3. You may have guessed that the IDLE (without the 3) icon opens IDLE for Python 2.

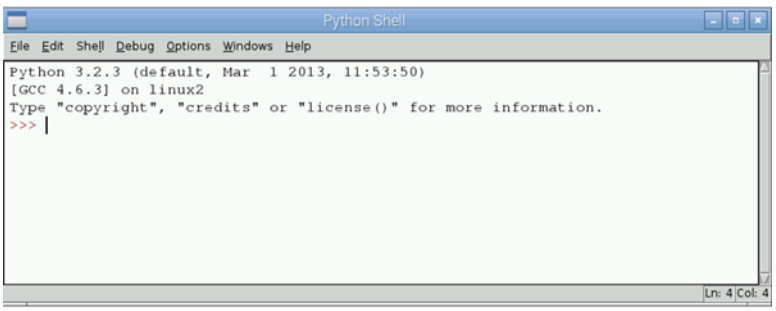


Figure 1.20 IDLE is a development environment that makes it easier to write Python programs. This is the IDLE Python Shell that you can use to enter Python commands or instructions one at a time.

NOTE To start the Python Shell from the Raspbian command line, type `python3` and press Enter. You'll see a `>>>` prompt and may interactively enter Python commands. When you're finished using the Python Shell, `type exit()` and press Enter to end your Python session.

The Python Shell shown in figure 1.20 allows you to enter Python commands and press Enter to execute them. The command prompt lets you type in commands after the triple greater-than symbols (`>>>`).

Do the following:

1. Enter `3 + 4`.
2. Press Enter.

The screen displays the answer: 7. Try some subtraction:

1. Enter `17 - 9`.
2. Press Enter.

The screen displays the answer: 8. Now let's make Python talk to you by printing a message to the screen:

1. Enter `print("I am alive!")`.
2. Press Enter.

Your screen should display "I am alive!"

Outstanding work! You wrote three lines of code. When you pressed Enter after each one, the Raspberry Pi's processor executed those commands and did what you asked. That is powerful!

Fruit Picker Extra: shopping at the Pi Store

Your Raspberry Pi can do many things. We've included special sections throughout the book called *Fruit Picker Extras* to teach you some different things your Pi can do. This Fruit Picker Extra is about shopping at the Pi Store.

The Pi Store is an online app store that provides access to games, apps, and resources for your Pi (see figure 1.21). You can browse the Pi Store from any device, such as a mobile phone or laptop. To access it from your Raspberry Pi, double-click the Pi Store icon on your desktop. If you want to download content to your Pi, you need to have your Pi connected to the internet, and you'll also need to create an IndieCity account with an email address and password.

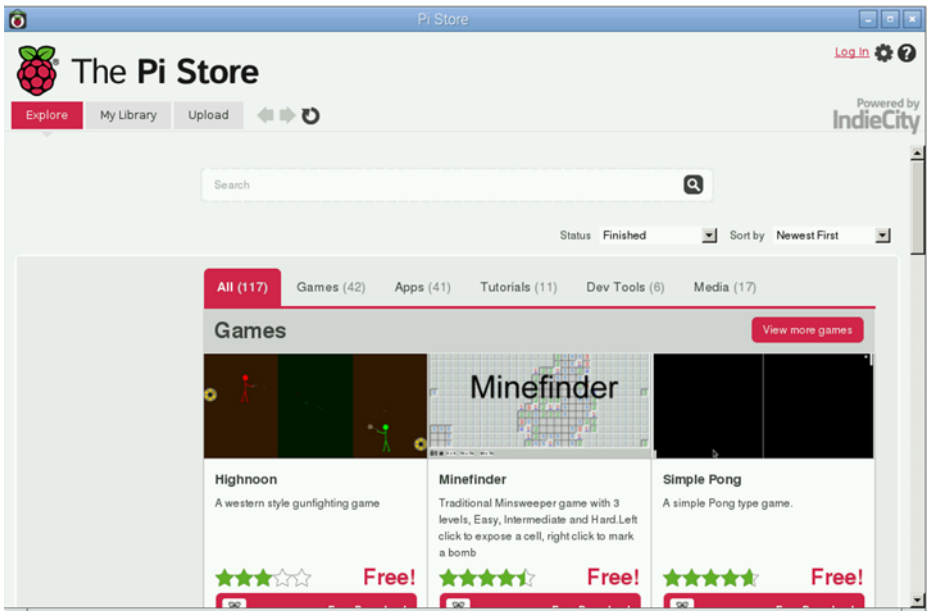


Figure 1.21 You can access the Pi Store from the icon on your Raspbian desktop. The store allows you to browse and download apps and content including games, tutorials, and digital magazines. You'll find free and fee-based content, organized into five categories: Games, Apps, Tutorials, Dev Tools, and Media.

Some apps are free; others require you to pay a fee. You'll find great resources, such as free issues of *MagPi,* the Raspberry Pi community magazine, a digital magazine full of tips, projects, and programming tutorials (look for these in the Pi Store's Media category). Have fun downloading free games and tutorials onto your Pi!

Challenge

Each chapter will have challenges at the end for you to try. If you can't figure them out, check the back of the book (see appendix C) for hints and answers.

Scavenger hunt

Time to explore your Raspberry Pi with a scavenger hunt. The goal is to learn more about the Pi by looking around, opening applications, and playing with them. Try to complete this list of scavenger-hunt items:

1. Find a game where squirrels eat other squirrels. Can you achieve the title of Omega Squirrel? Hint: Double-click the Python Games desktop icon to look for it.
2. Find a calculator application on your Raspberry Pi. Calculate the answer to a math problem: 87×34. Hint: The calculator is found under Menu > Accessories.
3. Without unplugging your Raspberry Pi, can you figure out how to shut down or restart it?
4. Turn your desktop's background black.
5. Bonus: Open Scratch, and try to make a cat dance.

Consider yourself an official Raspberry Pi explorer. If you want, take some more time to click some icons and see what they do. You've accomplished a lot!

Summary

The Raspberry Pi is like other computers in a lot of ways, but with several important differences. The similarities with other computers include these:

- A Pi requires a keyboard, mouse, and monitor, much like other desktop computers. The ports for plugging these in are part of the Pi.
- The Pi can be set up with a desktop OS, Raspbian. It's similar to Microsoft Windows or Apple OS X.
- Although its computing power is limited (similar to a smart phone), the Pi can still allow you to do many things you do on a desktop or laptop, such as browsing websites, playing games, and listening to music.

The Raspberry Pi has qualities and capabilities that make it special and unique. These key differences from other computers include the following:

- The Pi's cost and size are much smaller, making it a great candidate for projects.
- The Pi was designed for programming in Python and comes preloaded with the Python development environment so you can get coding right away.
- The Pi uses an SD memory card to store all files and software, including the OS.
- It has GPIO pins that can send and receive electrical signals. In part 3 of this book, you'll learn how you can use these to create projects that interact with the world around you.

2

Exploring Python

In this chapter, you'll learn how to interact with your Raspberry Pi by using Python to

- *Do math calculations quickly and easily*
- *Store information using variables*
- *Get messages to display on the screen*
- *Create and run your first program in Python*

An exciting part of programming is getting the computer to interact with you. It's the first step toward having the computer feel artificially intelligent.

Playing with Python

One of the best ways to learn to program is by exploring and playing. When you play, you try things and see what happens. You learn by experiencing the act of programming and seeing results. In this approach, you'll try entering different commands and see what happens.

Open IDLE for Python 3 by clicking the Menu button and selecting Programming > Python 3 on your Raspberry Pi's desktop (see figure 2.1). After you click it, you'll need to wait a few seconds while IDLE opens.

Figure 2.1 The Python 3 icon on your Raspberry Pi opens an interactive programming shell for Python 3.x.

NOTE There are both Python 3 and Python 2 icons under Menu > Programming on your desktop. Make sure you click Python 3 and not Python 2.

The Python 3 icon opens IDLE.[1] You'll see a prompt, ready for your commands—this is the Python Shell (see figure 2.2). With the Python Shell open, let's see how you can start talking to your Raspberry Pi using Python.

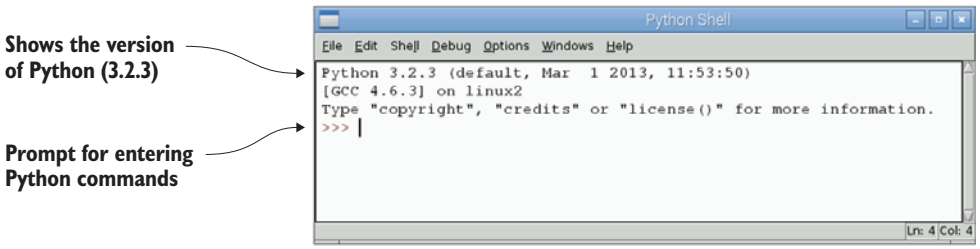


Figure 2.2 The Python 3 application under Menu > Programming on the Raspberry Pi desktop opens IDLE to the Python Shell for Python 3.x.

[1] The specific version of Python preinstalled on your Raspberry Pi may vary depending on when you purchased it. As of this writing, most Raspberry Pis come with Python version 3.2.3.

Discovering Python's mathematical operators

One of the core capabilities of a programming language is its ability to do math, or, in programmer-speak, to perform mathematical operations. Let's try different mathematical operations to see what works and what doesn't.

Adding and subtracting

Suppose you go to your favorite restaurant and order a burger, fries, and an orange soda. You want to know how much you owe. The menu (see figure 2.3) says the burger is $5.49, fries are $1.99, and the orange soda costs $1.49.

Figure 2.3 The menu at your favorite burger restaurant

Use Python to figure out the total. In the IDLE Python Shell, enter

```
>>> 5.49 + 1.99 + 1.49
```

Press Enter to see Python calculate the result: 8.97, or $8.97 (see figure 2.4).

Great news: you remember you have a coupon for $3.00 off, so let's calculate the total again. In the IDLE Python Shell, enter

```
>>> 8.97 - 3.00
```

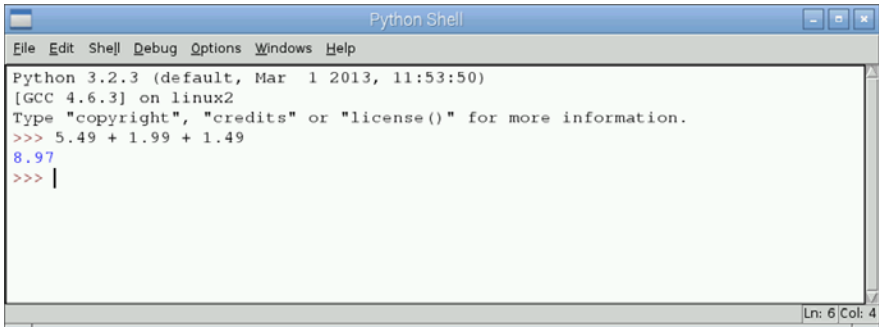


Figure 2.4 Use the + symbol to add numbers in Python.

The result is 5.970000000000001. Whoa! Why isn't it exactly 5.97? Well, it has to do with how computers store numbers as 1s and 0s. We aren't going to go over it here, but the footnote[2] has a web link where you can learn more. For now, the number is close enough for your calculations.

As you can see, Python is pretty good at doing math and uses familiar operators for addition and subtraction:

- The addition operator (+) calculates the sum of two numbers:

  ```
  >>> 4 + 5
  ```

  The result is 9.

- The subtraction operator (–) calculates the difference between two numbers:

  ```
  >>> 8 - 5
  ```

  The result is 3.

Python style: spacing of operators and numbers

Try entering 24 plus 32 without any spaces between the plus sign (+) and the numbers:

```
>>> 24+32
```

Then try it with lots of spaces:

```
>>> 24     +       32
```

Both result in the same answer: 56. When you're doing math, the number of spaces between the numbers and the operator doesn't matter. Python ignores the extra spaces and calculates the sum.

What's the best way? Well, Pythonistas (the name given to those who program in Python) believe that your code should be easy to read. The Python Style Guide[a] recommends using spaces before and after a mathematical operator. You don't have to, but it's easier to read!

[a] The Python Style Guide is referred to as PEP 8 and is found online here: www.python.org/dev/peps/pep-0008.

[2] Read more about decimal math (also called *floating-point math*) here: https://docs.python.org/3.4/tutorial/floatingpoint.html.

Let's see what other math you can do in Python.

> TIP When typing in large numbers, don't enter commas to separate groups of three digits. So 1,000 should be entered as 1000. Python can't interpret the comma separators in numbers, so you'll get some odd results if you add them. Python will interpret the commas as if you're typing in a list of numbers. For example, 12,231 is interpreted to be a list of two numbers: 12 and 231. You'll learn more about lists in part 2 of this book.

Multiplying and dividing

After scarfing down your burger, you find yourself hungry for two scoops of ice cream and a slice of raspberry pie for dessert. Ice cream is $1.79 per scoop, and pie is $3.50 per slice, so what is your total?

Use Python to figure it out. Try Python's multiplication operator (*):

```
>>> (2 * 1.79) + 3.50
```

You total bill is $7.08. You also see that you can use parentheses to group things.

Three of your friends join you at the restaurant, and each orders dessert. After more ice cream and pie, the total bill ends up being $33.36. They all agree to split the bill evenly. Use Python's division operator (/) to calculate the price they each should pay:

```
>>> 33.36 / 3
```

The result is $11.12 each. That's a lot of dessert!

With your belly full, you observe how you've seen Python perform multiplication and division and how you can use parentheses for grouping:

- The multiplication operator (*) gives you the product of two numbers:

  ```
  >>> 7 * 3.14
  ```

  The result is 21.98.

- The division operator (/) can divide two numbers:

  ```
  >>> 40 / 8
  ```

The result is 5.

- Parentheses can be used to group numbers so they're evaluated first:

```
>>> (3 + 7) * 10
```

Python answers 100.

What do you think this will result in?

```
>>> 3 + (7 * 10)
```

If you guessed 73, you're right. If you change the location of the parentheses, you'll get a different answer. We'll talk about this more when we examine the order of operations.

Figuring out whole numbers and remainders

Your friend mentions to you that there are 19,272 minutes of school remaining this year. How can you figure out how many hours and minutes? First you divide 19,272 by 60, because there are 60 minutes in an hour. You find that is 321 hours with a remainder of 12 minutes. In Python, you have two operators to give you the whole number and the remainder of a division sum:

- `//` (floor division) gives you the whole number:

```
>>> 19272 // 60
```

The result is 321.

- `%` (modulo) gives you the remainder:

```
>>> 19272 % 60
```

The result is 12.

You divided some large numbers, but let's look at how Python can handle even larger ones.

Exponents

An interesting fact you might've learned in Astronomy is that the Earth's distance to the Sun is approximately 1.496×10^8 km. Let's use Python to express this as a number. In Python you use the exponentiation operator (`**`) as follows:

```
>>> 1.496 * 10**8
```

Python answers 149600000.0 km.

Exponentiation lets you take two numbers (*a*, *b*) and raise one number to the power of the other(a^b). Python uses the exponentiation operator (`**`) between the two numbers (`a**b`) to do this. For example, if you wanted to raise 2 to the third power, you'd enter

```
>>> 2 ** 3
```

The result is 8 (`2 * 2 * 2 = 8`).

Try another:

```
>>> 122 ** 5
```

The result is 27,027,081,632 (`122 * 122 * 122 * 122 * 122 = 27027081632`).

> NOTE On older versions of Python, you may see `122**5` show the result 27027081632L. This is because previously Python added the letter `L` to denote really long integers.

Exponentiation can be useful if you're solving problems like these:

- Estimating astronomical distances
- Calculating bank account balances based on a given interest rate
- Predicting a population size for animal colonies based on a given growth rate

Square roots

You can figure out square roots by using an exponent of 1/2, or 0.5. This is the same as taking a square root:

```
>>> 14400**0.5
```

The result is 120.0.

Challenge: stacking Pis!

How many Raspberry Pis would need to be stacked end to end to reach the Sun? You can measure your Pi, and you'll find that a Raspberry Pi measures 85.6 millimeters or 0.0856 meters. First, you need to convert the Pi's measurements to kilometers by dividing 0.0856 by

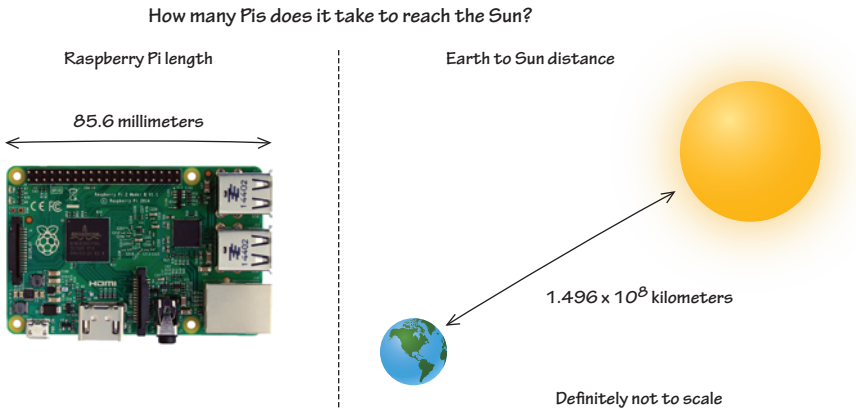


Figure 2.5 The distance from the Earth to the Sun is approximately 149,600,000 km. The Raspberry Pi is 85.6 mm in length.

1,000; then you divide the distance from the Earth to the Sun by the Pi's length in kilometers (see figure 2.5). This should give you the distance to the Sun, expressed as a number of Pis.

Enter the equation into Python:

```
>>> 1.496 * 10**8 / (0.0856 / 1000)
```

Python answers 17476635514O1.8694. That is more than 1.7 trillion Raspberry Pis stacked end to end. It's kind of fun to think about that many Pis!

Types of numbers: integers and floats

So far, you've used both integers and decimal numbers in your calculations. In Python, decimal numbers are also called *floating-point numbers*, or *floats* for short. Here are some examples of floats:

```
1.2
0.00001
3.14159
1000000.01
```

Checking types

Try entering this:

```
>>> type(3.14)
```

Python will answer you: `<class 'float'>`. You've just used Python's built-in tool for checking the type of something. These built-in tools are called *functions*. You'll see more of these later. Let's see what this does:

```
>>> type(10001)
```

Did you guess it? This returns `<class 'int'>`, where `int` stands for *integer*.

So far, you've typed in numbers and performed calculations. But if you want to change one number, you have to type all the information again. You also have no way of saving information—you have to look up and type the number each time. Good news! There is a better way.

Storing information using variables

There are times in programming when it's easier to store information than to type it in over and over again. *Variables* provide that special capability. Variables give you a way to store information and retrieve it anytime. Let's look at an example.

Imagine that you own a pizza restaurant, and your prices are shown in figure 2.6.

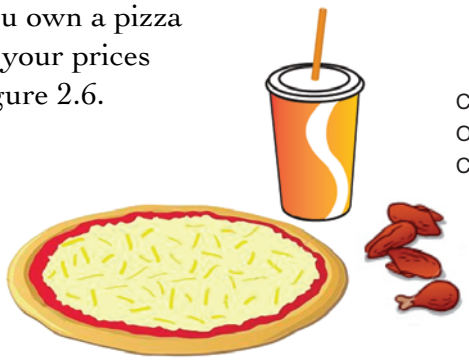


Figure 2.6 The menu at your pizza restaurant

The first customer, Daniel orders a meal of pizza and orange soda:

```
>>> 14 + 1.5
```

Daniel's meal costs $15.50.

A second customer, Erin orders pizza, orange soda, and wings:

```
>>> 14 + 1.5 + 8
```

Erin's meal costs $23.50.

Each time you want to calculate a meal's cost, you must remember or look up the price of each item and type it in. Imagine if you had a menu of 15 items and 100 customers. It would take forever to look up the items and add their prices together! You'd also be prone to making mistakes. Let's have the computer do this work for you.

Creating variables and assigning values

This is a perfect place to use variables in a program. Variables store information to make your life easier. (We're programmers, and we like to be lazy. At least we're always trying to find a more efficient way to do things.) Let's do this again but create variables for each of the food items. The first step is to define your first variable and set its value:

```
>>> cheese_pizza = 14
```

Let's take a close look at how this code works in figure 2.7.

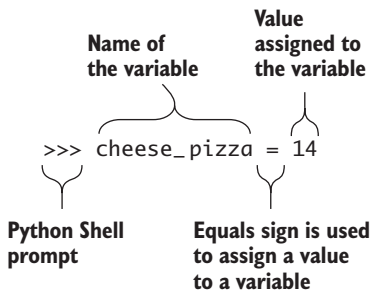


Figure 2.7 A variable stores information and can be created and assigned a value. The equals sign is used as an operator (also known as the *assignment operator*) between the name of the variable on the left and the value assigned to it on the right.

Next, let's create the other two variables for orange soda and wings:

```
>>> orange_soda = 1.5
>>> wings = 8.00
```

Nothing is displayed on the screen after you enter each line, but Python stores the variables and their values in the memory of your Raspberry Pi.

Displaying variable values

How can you check what's stored in a variable? Like the `type` function earlier, you use another built-in function in Python called `print`, like this:

```
>>> print(cheese_pizza)
    14
```

Print doesn't mean to print something with paper and ink. In Python, printing means to display something on the screen.

> NOTE When you're working in the Shell, Python displays the result of expressions. But if you assign a sum to a variable, the Shell doesn't show the value unless you use `print`.

Using `print`, you've seen that `cheese_pizza` has the value 14 stored in it. You should feel confident that your variables are holding the information you put in them.

Let's see if you can use variables to figure out a meal cost (without having to look up numbers):

```
>>> meal_cost = cheese_pizza + orange_soda
```

Print `meal_cost` to see its value:

```
>>> print(meal_cost)
15.5
```

Python displays 15.5. Now, let's calculate the cost of the other meal:

```
>>> meal_cost = cheese_pizza + orange_soda + wings
>>> print(meal_cost)
23.5
```

Python answers 23.5. The more calculations you need to repeat, the more you'll appreciate how variables can save you time and effort. Congratulations—you're using variables to store information!

> DEFINITION The process of putting a value into a variable is called *assignment*.

Before you start creating a lot of variables, let's learn the guidelines for naming them.

NAMING VARIABLES

Everyone has had the problem of not being able to read someone else's handwriting. The writer might know what they wrote, but you're unable to decipher it. You want to avoid this same confusion with variables. In order to do that, there is a set of guidelines for creating clear

variable names—names that make sense to you and to someone else reading your code:

- Don't use any spaces. Instead, use an underscore (_).
- The Python Style Guide recommends using lowercase and underscores between words to make your code easy to read.
- Don't start with a number.
- Don't use any of Python's reserved words for your variable name (see the sidebar "Watch out for reserved words").

Here are some examples of variable names:

```
>>> shoe_size = 10
>>> age = 16
>>> favorite_color = 'blue'
>>> first_name = "John"
>>> pizza_slices_eaten = 4
```

Do your best to use meaningful variable names.

Watch out for reserved words

Certain words in Python are *reserved* because they're part of the Python language. You can't use these words as names for variables:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Python 3.x reserved words are used by Python for special purposes and may not be used for variable names.

ASSIGNING VALUES: THE LEFT SIDE AND RIGHT SIDE

When you're creating a variable and assigning it a value, put the name of your variable on the left side of an equals sign. Put the value you want to set it to on the right side of the equals sign. Let's create a variable `name` and set it to "King Arthur":

```
>>> name = "King Arthur"
```

In this line, the left side creates a variable called `name`, and the set of characters "King Arthur" is stored in it. Let's learn more about storing text in variables.

Storing strings in variables

Life isn't only about numbers. You may want to create programs that display absurd messages or tell a story on the screen. These messages are a type of data called *strings*. A string is a group of characters.

STRINGS

Python gives you the ability to store a group of characters (or strings) in variables. You've already used strings in the example with "King Arthur".

Here are some things you should know about strings:

- They always must start and end with quotation marks.
- You may use either single quotes ('Hi') or double quotes ("Hi"), but you can't mix them ("Hi'):

  ```
  >>> message = "Greetings Earthlings"
  ```

  Or, in single quotes:

  ```
  >>> message = 'Greetings Earthlings'
  ```

- When a number is placed inside quotation marks, it's a string.
- Strings can be short (zero or only a few characters) or many characters long.
- Strings can even be empty. These are called *zero-length strings*:

  ```
  my_string = ""
  ```

EXAMPLES OF STRINGS

Some examples of strings will give you an idea of what's possible:

```
"Y"
"No"
"Spam"
"Yeah, remarkable bird the Norwegian Blue"
```

```
"There he is!"
"No, no sir, it's not dead. It's resting."
"17"
"RUNAWAY, RUNAWAY, RUNAWAY!"
"Tuesday"
```

MEASURING THE LENGTH OF A STRING

You can use the `len` function to have Python tell you the length of a string. We'll talk more about string functions in chapter 3, but here is an example of using `len`:

```
>>> your_nickname = "Pi Master"
>>> len(your_nickname)
9
```

Or try a longer one:

```
>>> quote = "To be, or not to be, that is the question."
>>> len(quote)
42
```

Even the spaces are counted when determining the length of a string. This is a great point to talk about spaces.

SPACES COUNT

Although spaces may seem like nothing, they're considered characters. You can create strings that are a single space or set of spaces, such as

```
short_set_of_spaces = " "
long_set_of_spaces = "    "
```

You now know about variables and about strings, a type of data that can be stored in them. Let's see how you can vary your variables.

Changing the value of variables

As you may have guessed already, the value stored in a variable can be changed or updated. Try it. You're making up a password for your computer. Create a variable `password`, and set it to `bunny`:

```
>>> password = "bunny"
```

Now let's change the password to `dragon`:

```
>>> password = "dragon"
```

What value do you think is stored in `password`: "bunny" or "dragon"? Let's check the value using the `print` function:

```
>>> print(password)
dragon
```

The value `dragon` is displayed. Notice how Python replaces the value stored in the variable when you assign it a new value.

VISUALIZING VARIABLES AS BOXES

A way to visualize this is to imagine that creating a variable is like making a box—a box for storing information. When you create the box, you give it a name and store a value in it. Figure 2.8 is a graphical depiction of creating a variable and reassigning a value to it.

Changing the value of a variable is easy to do in Python. Let's look at another example.

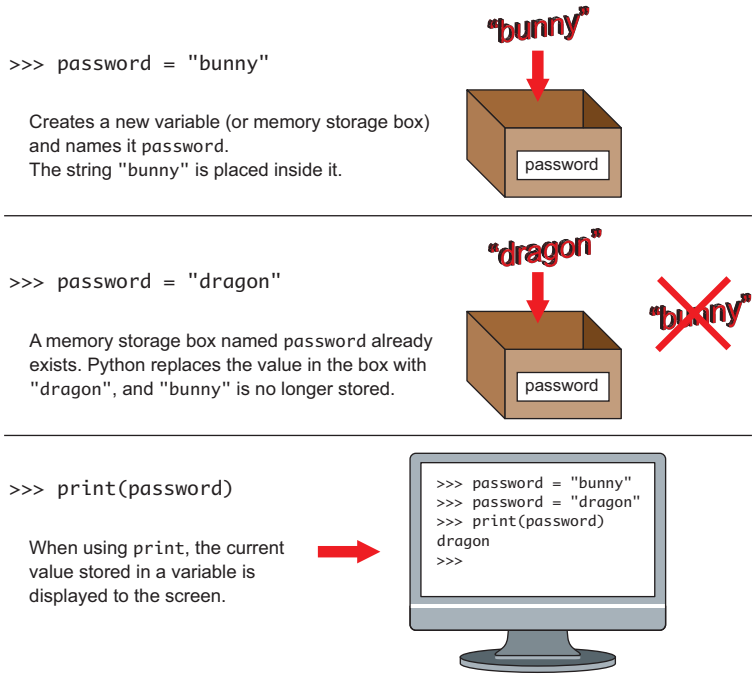


Figure 2.8 When a variable is created, it's stored in your Raspberry Pi's memory. You can change the value of a variable at any time. Using the `print` function, you can display the variable's value on the screen.

VARIABLE REASSIGNMENT

Let's see how Python evaluates these statements:

```
>>> x = 10
```

This sets `x` equal to the value 10. Next, you do a calculation with `x` and store the result of the calculation in `x`:

```
>>> x =  x * 10 + 32
```

When Python evaluates this line, it first tackles the right side of the equals sign:

1. Python evaluates the right side of the equation: `x * 10 + 32`.
2. Python retrieves the current value of `x`, 10, and calculates `10 * 10`.
3. Python adds 32 to this amount. The right side of the equals sign is 132.
4. It does the left side of the equals sign last. The result, 132, is stored into the variable on the left side of the equals sign: `x`.

You've seen how Python can store and retrieve information using variables. Variables save you time because they hold the value they're given, meaning you don't have to remember values or look them up. Variables can take the form of numbers or strings, and you can check the value stored in a variable using the `print` function.

Excellent! You've seen how the order for variable assignment is important. Check out how the order of math operations matters.

ORDER OF OPERATIONS

What do you think Python will return if you enter the following?

```
>>> (3 * 2) * 5**3 / 25 + 10
```

If you guessed 40, you're correct. Python follows the order of operations that you learned in math class.

> TIP You may recall BOMDAS or PEMDAS from school. This pattern of letters is useful for remembering the order you should evaluate operations in a math equation. Python follows this same order of operations: Brackets (or Parentheses), Orders (or Exponents), Multiplication and Division, and then Addition and Subtraction.

First it evaluates anything grouped in parentheses or brackets. `3 * 2` is equal to 6. Let's replace the `3 * 2` with `6` and go to the next step:

```
>>> 6 * 5**3 / 25 + 10
```

The exponents (or orders) are analyzed next. `5**3` is 125 (the same as `5 * 5 * 5`):

```
>>> 6 * 125 / 25 + 10
```

Multiplication and division come next, and you work from the left to the right. `6 * 125` is `750`. `750 / 25` is 30:

```
>>> 30 + 10
```

The final step is addition and subtraction. `30 + 10` is 40. Graphically, figure 2.9 shows the order in which the example equation is solved in math and how Python does it.

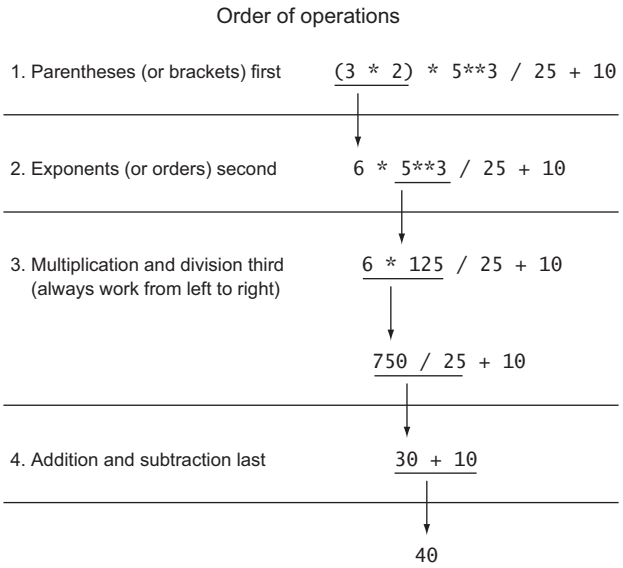


Figure 2.9 Python follows the order of operations used in mathematics. You may know it as BOMDAS or PEMDAS: Brackets (or Parentheses), Orders (or Exponents), Multiplication and Division, and finally Addition and Subtraction.

You're pretty good at doing math in Python. You're ready to learn more about using Python to communicate and display text on the screen.

Displaying text on a screen

It's fun to interact with technology and have it respond. This can take the form of playful responses by a computer, making it feel more human. Or computer responses can be more practical, displaying personal data on a website form. In either case, you want your computer to communicate with you.

Displaying text on the screen, also referred to as *printing* in Python, is a direct way for a computer to communicate with you. You can use printing to have your Raspberry Pi do things like this:

- Show random, silly messages.
- Describe spooky scenes as part of an adventure game.
- Spit out the answers to complex math problems.

Printing to the screen is a key way to output all kinds of information.

Using the print function

Earlier in this chapter, you used the `print` function to display the value of variables. Let's go over more about using the `print` function. Try printing the message "Hello World!" to the screen like this:

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

Take a closer look at how you can use the `print` function in figure 2.10. Python prints "Hello World!" to the Python Shell.

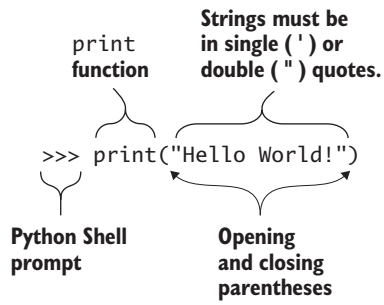


Figure 2.10 The `print` function in Python displays text on the screen. The string inside the parentheses must be enclosed in single or double quotation marks.

REPEATING TEXT

Let's try something a bit different. Type in

```
>>> message = "Hello, I am your Raspberry Pi!"
>>> print(message)
```

This prints the message on the screen once. You can use the multiplication operator with a string to print it many times:

```
>>> print(message * 100)
```

The message cascades across and down the screen 100 times (see figure 2.11).

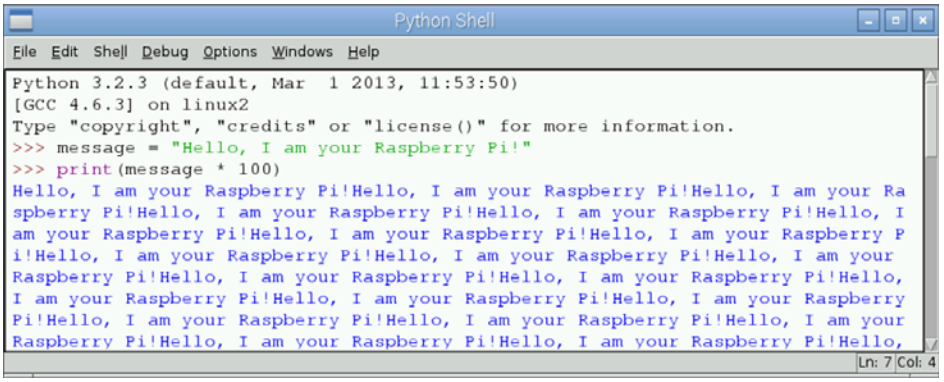


Figure 2.11 The Python `print` function can display text on the screen repeatedly if you use it with a string and the multiplication operator (*).

Have fun with this. Try some bigger numbers and different messages to see what you get.

Troubleshooting

We're all human, so things can go wrong when we're pressing keys and typing in code. A common error you might make when creating a variable that is storing a string is forgetting to close your quotation marks:

```
>>> message = "Hello, I am your Raspberry Pi!
```

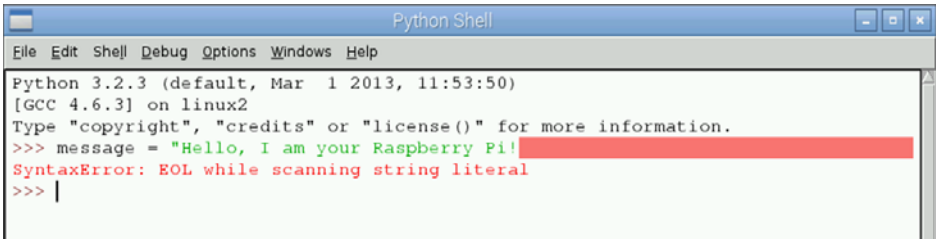


Figure 2.12 Remember to place quotation marks at the beginning and end of your strings. If you forget to close your quotation mark, Python will display an error.

Notice that the quotation mark after the exclamation point is missing. It may sound goofy, but think of quotation marks as hugs. When you hug someone, you wrap your arms around them. One quotation mark must go on either side of a string to complete it. If you ran this code in the Python Shell, you would receive an error, as shown in figure 2.12.

Python displays an error message (`SyntaxError: EOL while scanning string literal`). You can fix it by typing the string again with both the opening and closing quotation marks around it.

Creating programs

Imagine again that you own a pizza shop and you want to use Python to calculate the cost of a meal, including tax. A customer orders a meal of two slices of pizza and orange soda. Let's start by creating two variables with the menu prices:

```
>>> pizza_slice = 3.5
>>> orange_soda = 1.50
```

Create two variables to keep track of the number of slices and number of drinks:

```
>>> num_slices = 2
>>> num_drinks = 1
```

Next let's calculate the cost of the meal without tax:

```
>>> meal_no_tax = (num_slices * pizza_slice) + (num_drinks *
  orange_soda)
```

Define the tax rate of 5%, and figure out the tax:

```
>>> tax = 0.05
>>> meal_cost = meal_no_tax + (meal_no_tax * tax)
>>> print(meal_cost)
8.925
```

Now imagine if one or more of the numbers changed. Let's say pizza slices are now $4.75 and orange soda is $1.75. You'd have to enter all the information again. That takes way too long.

A better way is to put the eight statements into a text file. Then you can tell Python to read the file and execute the instructions.

> DEFINITION A *program* is a set of instructions. Python programs can be created in a text file. The programs can be run (or executed) over and over again.

Now you can run the program again and again, making updates whenever needed. If the cost of menu items changes or a customer wants a different number of slices, you can update the program and run it again. That is a big time-saver!

A computer program is a set of instructions. So far, you've used the Python Shell to type in commands one at a time. Programs allow you to create, save, and run more complex sets of instructions. You can easily edit your programs and run them again. Your programs might be as short as a few lines, or thousands of lines long.

Writing Python programs with IDLE

To write a program, you need a way to input the instructions. IDLE will be your program of choice for this. IDLE is an application that makes it easier to develop programs.

A SPELL CHECKER FOR PYTHON

If you've ever used Microsoft Word or Gmail, you're familiar with the spell-checker feature. It's saved thousands of homework assignments from receiving low grades and stopped misspelled emails from being sent. Each program highlights words you misspell, so you can easily find them and make corrections.

When you write programs, you want something to help catch your mistakes. IDLE does that for you. IDLE automatically color-codes your Python statements to let you know you're using the correct spelling. By using color-coding, IDLE can help alert you if you enter a command incorrectly and highlight errors or bugs. In later chapters, I'll introduce you to some of the features of IDLE.

INTEGRATED DEVELOPMENT ENVIRONMENTS

Other programming languages have software applications similar to IDLE that make the process of programming more enjoyable, help prevent errors, and even suggest fixes. As a group, these software applications are called *integrated development environments (IDEs)*. IDLE is one of the most popular ones for Python.

USING TEXT EDITORS

In addition to IDLE, you can write and save Python programs in any text editor you like. For example, you could use Leafpad or Nano, which are other simple text editors that come with Raspbian. A word of caution: they allow you to write, but they don't help you avoid errors or find mistakes in your code, making IDLE a better choice.

Starting a new program

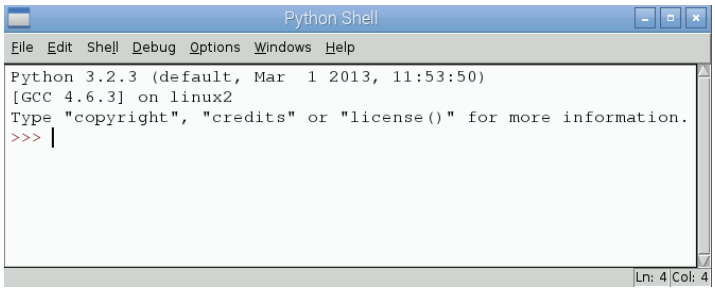
Let's create our first program. While using the IDLE Python Shell, select File > New Window. You'll see a blank new window appear, with the title Untitled at the top (see figure 2.13). This is the IDLE text editor.

> **TIP** The keyboard shortcut to open a new IDLE text editor window is Ctrl-N.

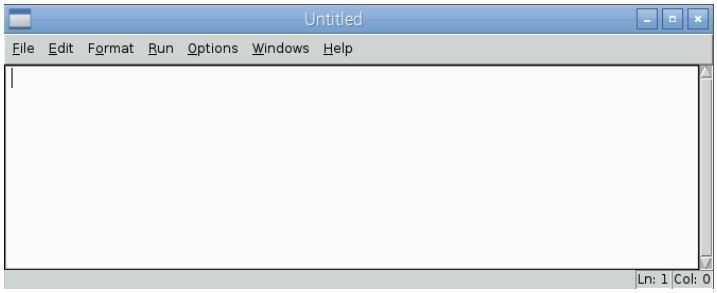
Let's write a program in the IDLE text editor. Enter the following lines of text:

```
message = "And now for something completely different."
print(message)
```

> **TIP** The text editor automatically highlights keywords in the Python language. In this example, you'll notice `print` appears in purple text, signifying it's a Python keyword. Strings are color-coded green.



IDLE Python Shell



IDLE text editor

Figure 2.13 The top window is the IDLE Python Shell. The bottom window is the IDLE text editor that can be used to create and edit Python programs. You can open the IDLE text editor using Ctrl-N or by selecting File > New Window from the IDLE Python Shell.

This is a classic line from *Monty Python's Flying Circus*. The show begins with this quote. Figure 2.14 shows the program in the IDLE text editor.

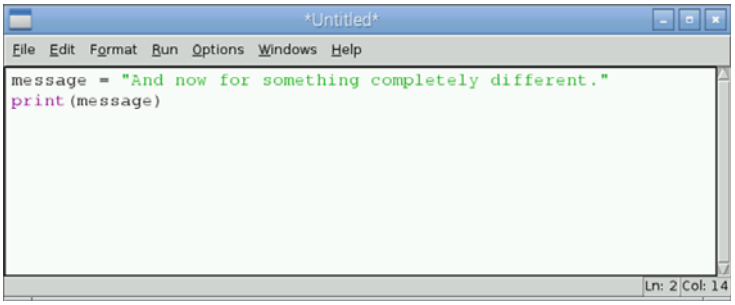


Figure 2.14 IDLE provides a text editor that helps you write Python programs. The editor highlights words to help you compose your programs and identify errors. This program prints a message to the screen.

Now that you've written a program, you'll want to save it so you can open it, run it, and edit it later.

Saving programs

To save the program, choose File > Save. A Save dialog appears. Name the file FirstProgram, and click Save (see figure 2.15). By default, the file will be saved to your /home/pi folder. If you want, you can create a folder for your Python programs.

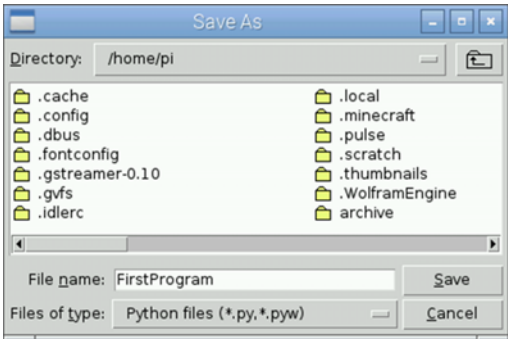


Figure 2.15 Save programs in IDLE using the File > Save menu selection or by pressing Ctrl-S. The default save location is /home/pi. When the file is saved, it has .py appended to the end of its name, signifying that it's a Python program.

TIP The keyboard shortcut to save a program is Ctrl-S.

NOTE When you click Save, the program is saved to your /home/pi folder with the extension .py. You can use File Manager to open your /home/pi folder and see the file you've saved: FirstProgram.py.

While using the Python text editor, you can run the program by clicking Run > Run Module, or you can press F5. When you do this, the IDLE Python Shell becomes the active window, and you'll see the message printed to the Shell (see figure 2.16).

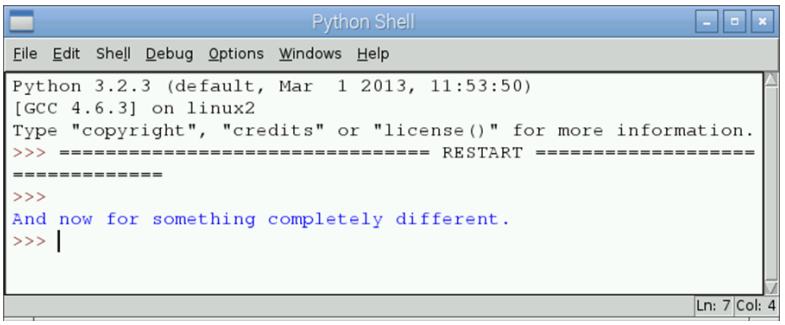


Figure 2.16 You can run programs from the IDLE text editor. Running a program in IDLE displays the results of the program in the Python Shell. This shows the output of your first program by displaying a message.

Python interpreting the program

When you run your program, Python opens the file and interprets each line of text. The first line creates a variable `message` with the stored value "`And now for something completely different`." The second line of your program calls Python's `print` function and passes it the variable `message` to output to the screen. Excellent—you'll continue to build more programs in the next part of the book.

Fruit Picker Extra: creating documents

This special section is about teaching you new and different things your Pi can do. This extra is about creating documents.

Writing silly things and saving them

Let's start by creating a simple text file and saving it. Using a Raspberry Pi to do homework can be a lot of fun. Maybe you'll write a document describing your latest idea for a game or create a collection of short stories. Rather than use your parent's computer or a pen and paper, use your Raspberry Pi.

Luckily, Raspbian comes with an application called Leafpad. It's a lightweight software program for creating documents with text.

CREATING A TEXT FILE IN LEAFPAD

Here are the simple steps for creating a document in Leafpad:

1. Click the Menu button in the upper-left corner of the desktop.
2. Hover over Accessories.
3. Find Text Editor, and click it. This opens Leafpad.
4. Type in the Leafpad window: `I'm a lumberjack and I'm okay!` (see figure 2.17).

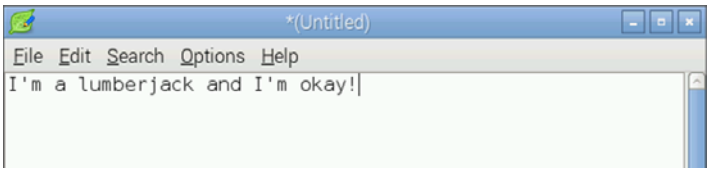


Figure 2.17 Leafpad is a text editor that comes with Raspbian. You can access Leafpad from the Accessories menu.

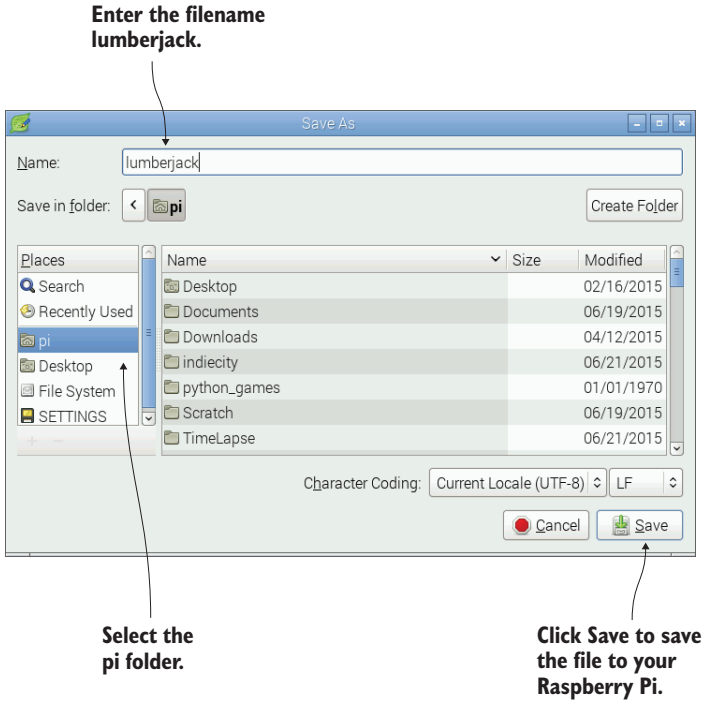


Figure 2.18 Saving a file in Leafpad lets you choose the folder to save to and enter a filename. The Save window works similarly to how you might save a file in Microsoft Word.

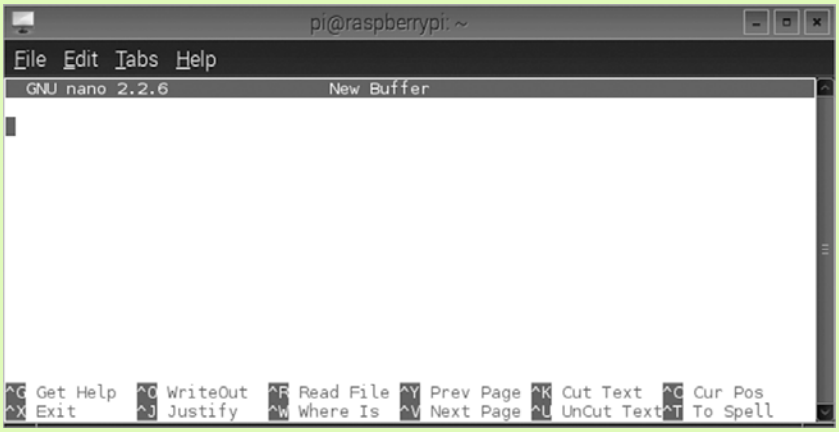
Now that you've created your file, let's save it (see figure 2.18):

1. Select File > Save, or use the keyboard shortcut Ctrl-S.
2. A window appears that you can use to save your file. You need to pick the folder you want to save your file in. Click the folder labeled pi. This is your personal folder where you can save your files.
3. In the Name box, enter `lumberjack` for the filename.
4. Click the Save button.

Congratulations! You saved the lumberjack file to your Raspberry Pi's memory card in the folder located here: \home\pi (this means the file is saved in the home folder and in a subfolder called pi). The file contains

Nano: a command-line text editor

Leafpad uses windows and is therefore only available from the Raspbian GUI. But if you decide you prefer to use the Raspbian command line, there is a handy text editor called nano that you can use. Type nano in the command line and press Enter to open nano. Nano uses keyboard controls to open, save, and close files. Here is an example of the nano text editor:



You must use the keyboard, not the mouse, to make selections and perform actions in nano. For example, Ctrl-X exits nano. Once you get used to using the command keys to get around, nano is useful if you decide you prefer using Raspbian in command-line mode.

the sentence you typed: "I'm a lumberjack and I'm okay!". Go ahead and close Leafpad.

FINDING A SAVED FILE

You saved the file. Now let's see if you can use File Manager to find it and open it again:

1. Open File Manager.
2. Click the folder icon on the left, labeled pi.

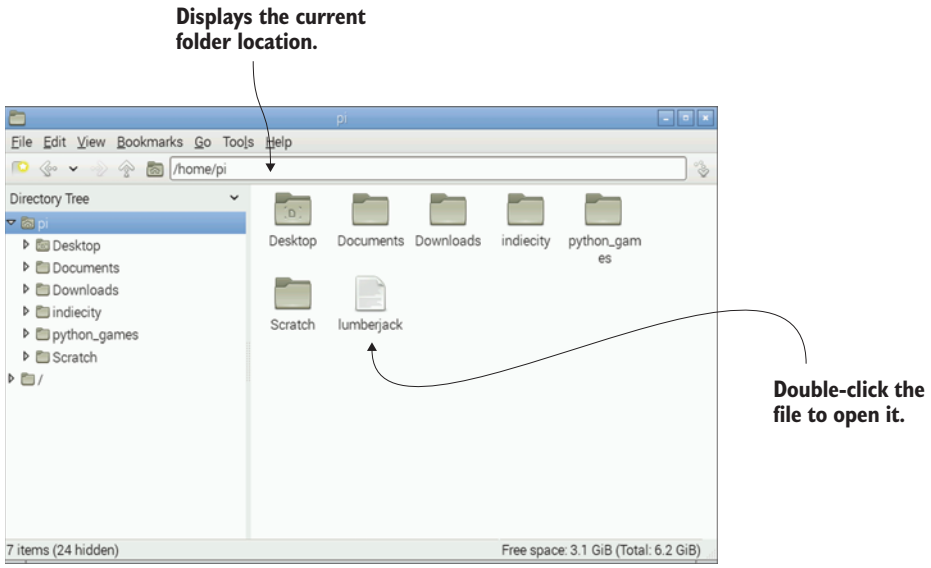


Figure 2.19 Viewing the contents of folders using File Manager

3. Look at the folders and files listed in the window. Notice at the top that the pi folder is located at \home\pi. This means the pi folder is located in the folder home on your Raspberry Pi's SD card.
4. Find the lumberjack file in the list of files, and double-click it (see figure 2.19).

Leafpad will open, and you'll see the message you typed. Now let's close Leafpad and learn how to enter some code.

Have fun making documents and exploring other things your Pi can do!

Challenges

Try these challenges, which will test your use of mathematical operators, printing, and variables.

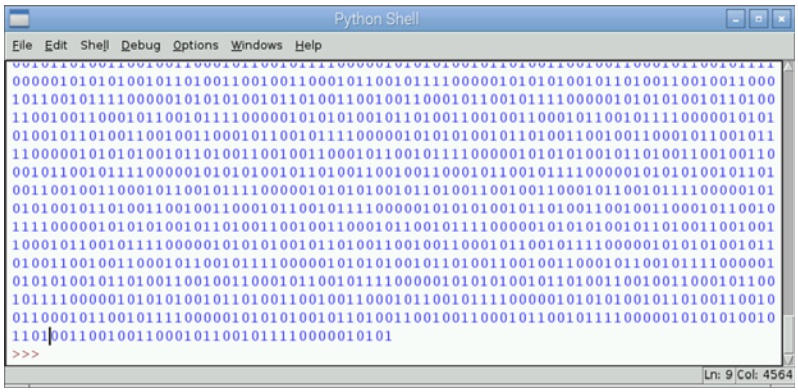


Figure 2.20 Try using the `print` function and strings to make a screen full of 1s and 0s.

The matrix

Using the `print` function, create a cascading screen of 1s and 0s as seen in popular computer graphics. Hint: remember how you used `print(message * 100)` to display a message 100 times on the screen. Figure 2.20 shows an example of what this might look like.

The matrix challenge is about creating a full screen of digits. Experiment with other numbers and characters.

Building a brick wall

For this challenge, create a variable named `brick` and store a string in it that, when printed over and over again, will make your screen turn into a brick wall (see figure 2.21).

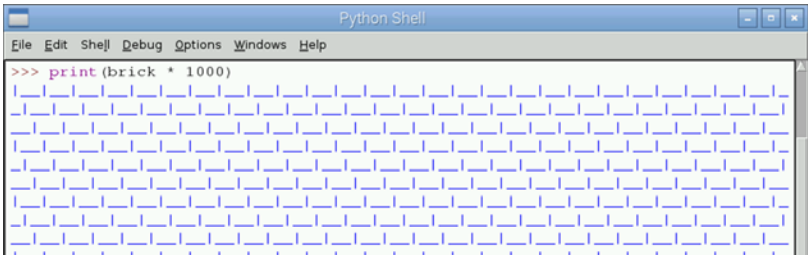


Figure 2.21 This challenge uses the `print` function and a string named `brick` to create a brick wall pattern on the screen.

Your goal is to figure out what string should be stored in the variable named `brick` to make this display. Good luck! Bonus: can you make your bricks look more like raspberries or have them contain the initials RPi for Raspberry Pi?

Pi electrons

For this advanced challenge, let's examine the electrical current flowing into your Raspberry Pi from the power supply. Then, let's see if you can express that electrical current in terms of the equivalent number of electrons flowing into your Pi per second.

> TIP You may have learned that electrical current is a measure of charge flowing past a point. One *amp* (or ampere) of current is equal to one *coulomb* of charge flowing each second.

The amount of current your Pi uses depends on how many USB ports you're using, but let's assume your Pi is using one amp. One amp is equivalent to the flow of 1 coulomb of electrical charge flowing per second. A single electron has the charge of 1.60×10^{-19} coulombs (or 0.00000000000000000160 coulombs). How many electrons per second does it take to equal 1 amp flowing into your Raspberry Pi? Hint: You can represent the charge of an electron as `1.60 * 10**-19`.

For hints and solutions to the challenges, see appendix C.

Summary

Programming is about being able to interact and communicate with a computer. Your Raspberry Pi comes with IDLE, a development environment for programming in Python. Python provides two different ways you can program:

- Interactively, by entering commands one at a time using the Python Shell. The Shell is useful for quick calculations or testing a command.
- By creating programs, or sets of commands, saved in a file. Programs allow you to write, edit, and run your code over and over again.

One of the first conversations you can have with your Raspberry Pi is to use Python to talk math. Python provides a full set of mathematical

operators you can use. Mathematical operators are handy when you need to perform calculations in your programs, such as keeping track of a player's position on the screen. Another way to interact is to use Python's built-in `print` function to display text to your Pi's screen. This lets you create programs that communicate between the computer and you.

An important idea in programming is using variables to store information—they save you time and can be used again and again. In Python, variables can store different types of data, including integers, floats (decimals), and strings. Using variables, you can store information and retrieve it any time. This is a key advantage, because it means you don't have to remember values; Python does it for you. You can also change a variable's value, which is a useful feature when you want to run the same instructions with different inputs.

Part 2

Playing with Python

Minecraft, Pac-Man, and Super Mario Brothers are great games, and they were all created by programmers like you. You'll have to gain more skills to make games like those, but you can create some basic games pretty quickly. All these games have the game player interact with the computer. The computer is programmed with *logic*: instructions that control how the game reacts to the player's choices. The game is constantly responding to input from the user, whether it is a button press or a key press.

Games are a good way to learn programming because they combine creativity, fun, and logical thinking into one project. Games are also interactive, requiring the user to make choices and the computer to respond to those choices. The goal is to make the game entertaining, so you'll use your creativity and imagination to add magic to your games. You decide how you want to program your game and how it responds!

In part 2, you'll build your own interactive games using Python and your Raspberry Pi. You'll start in chapter 3 by making a program that creates ridiculous sentences. You'll learn to use Python to ask users to enter information, store the information in variables, and make your Pi respond. Chapter 4 dives into how you can create a guessing game that makes your Pi more intelligent: it will make simple decisions based on the player's choices. You'll also see how to use Python to make your Raspberry Pi repeat some instructions over and over again. In chapter

5, you'll don a helmet and headlamp and descend into an underground cave. You'll create a text-based game where the player can choose where to go; based on their choices, they may find riches or face an untimely demise.

3

Silly Sentence Generator 3000: creating interactive programs

In this chapter, you'll see how you can use Python to

- *Create a welcome message for a game*
- *Add notes to your code*
- *Ask users to input (or type in) information and save it using variables*
- *Join strings*
- *Display information back to the user based on that information*

Visit a website, start up a game system, or open a mobile application, and it will probably ask you to enter a name and email address and create a password. These are all computer programs, and once you're logged in, they may display special messages at the top of the screen saying things like "Welcome, Aaron" (or whatever your name is). Some programs are very sophisticated, remembering the games you've played, the badges you've earned, the balance in your account, or the products you've viewed.

iTunes, Netflix, Facebook, and Gmail are all sites that use computer programs that ask you for information, save information, and interact with you based on that information. In this chapter, you'll see how to do this with Python by creating a ridiculously fun word game called Silly Sentence Generator 3000.

Creating a welcome message

In Silly Sentence Generator 3000, the game player (that'll be you) is asked to enter words such as nouns, verbs, adjectives, and so on. You'll store the words as variables and then use them to create ridiculous, nonsensical sentences.[1] Figure 3.1 shows an example of what the finished program looks like.

Think about the program like a machine that takes a set of inputs and then creates an output. You're going to put together the machine by creating the instructions that drive it. Conceptually, this "machine"

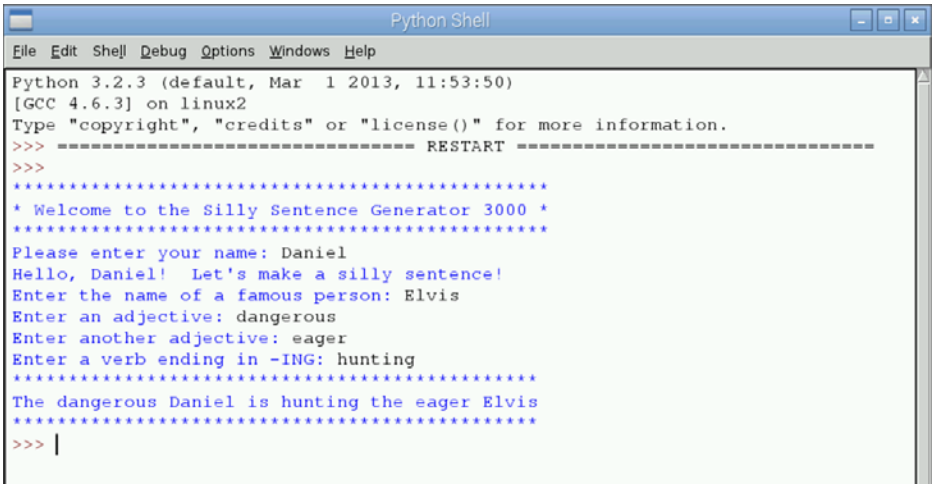


Figure 3.1 Silly Sentence Generator 3000 asks the user to enter their name and some words, and then it creates a silly sentence from those words.

[1] This is similar to the game Mad Libs, if you've ever played it.

might look something like figure 3.2. Change the words you put in, and you'll get a completely different result. That's part of what makes games so much fun!

Let's see how to create this game. Once you create it, you can change it and add to it however you like.

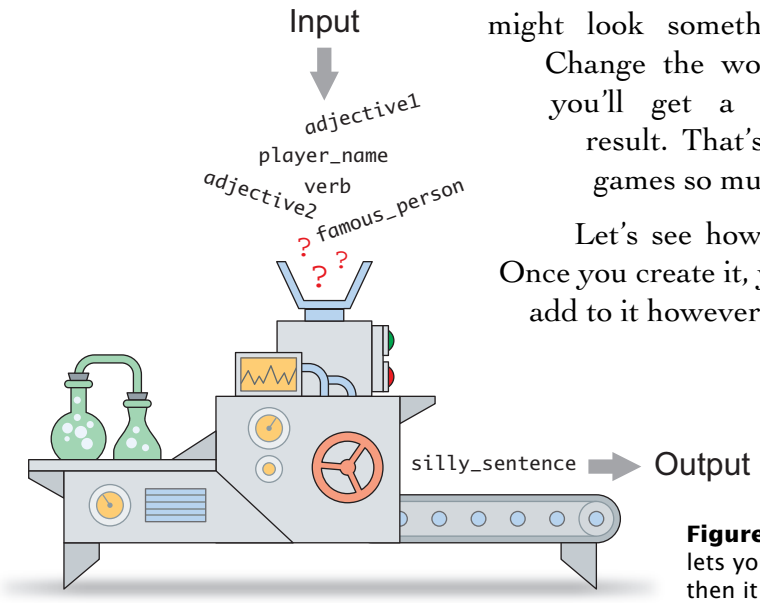


Figure 3.2 An interactive game lets you put in information, and then it creates an output.

Starting a new program

If you open a game, one of the first things you see is a main menu or title screen. Let's use what you know about displaying text on the screen to make your program display a title for your game. You start by opening IDLE and creating a new program. Open IDLE for Python 3 by clicking the Menu button and selecting Programming > Python 3 on your Raspberry Pi's desktop (see figure 3.3).

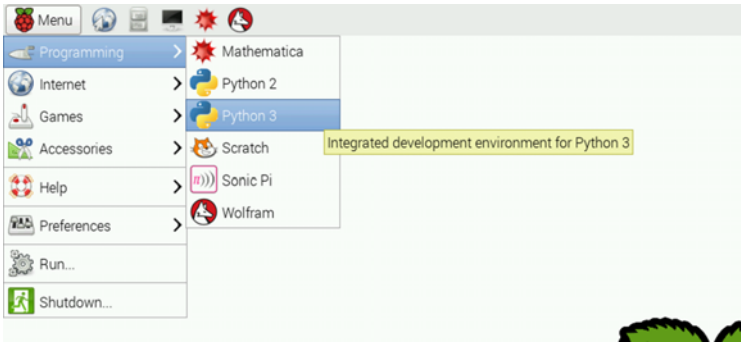


Figure 3.3 Select Menu-->Programming-->Python 3 to open the Python Shell on your Raspberry Pi.

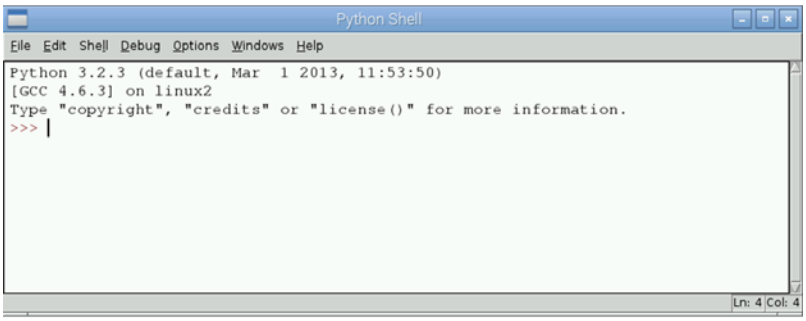


Figure 3.4 The Python Shell

Give your Raspberry Pi a few seconds to open IDLE. After IDLE opens, you'll see the Python Shell (see figure 3.4).

Press Ctrl-N or choose File > New Window to open the IDLE text editor. You'll see a blank window, ready for you to start typing in your program (see figure 3.5).

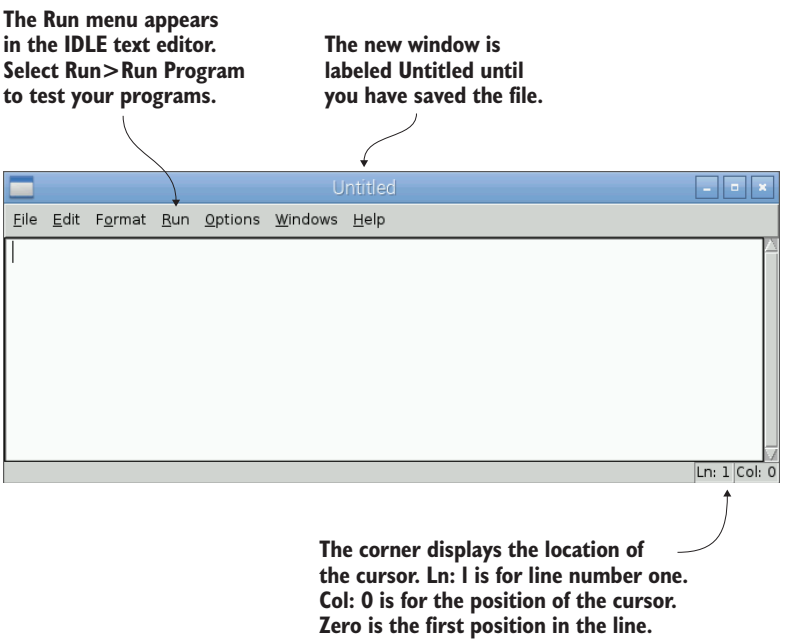


Figure 3.5 The IDLE text editor is where you can type in your Python program. You can also edit, save, and run programs using the menu options.

Using the `print` function you learned about in chapter 2, let's make a title screen:

```
print("*" * 48)
print("* Welcome to the Silly Sentence Generator 3000 *")
print("*" * 48)
```

Excellent. Feel free to elaborate on the welcome message and the artwork with different characters. Before you go much further, you should save the program.

Saving the program

Save the program by selecting File > Save or pressing Ctrl-S. This will open a window asking where you want to save the program and what to name it. Let's name it SillySentence (see figure 3.6). By default, IDLE saves your file to your /home/pi folder. Let's use that folder.

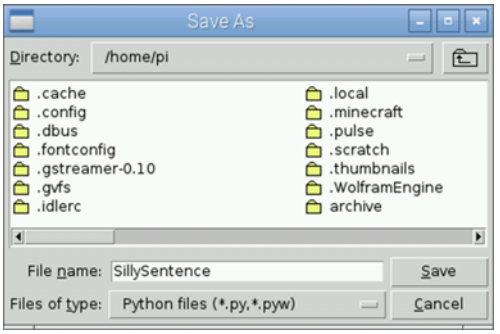


Figure 3.6 Save your file as SillySentence. This stores the file on your Raspberry Pi in your /home/pi folder so you can run the program and make changes to it.

Click Save, and the file will be saved as SillySentence.py (the .py file extension is automatically appended by IDLE). After you save the file, the title at the top of the text editor window will show the filename and file location, as you can see in figure 3.7.

The window title updates after you save. The title changes from Untitled to SillySentence.py.

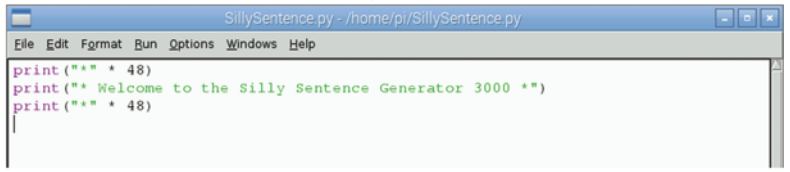


Figure 3.7 The first three lines of your program use the `print` function to create a welcome message for the Silly Sentence Generator 3000 program.

Guess the output. What do you think you'll get when you run the program?

Let's try it. Click Run > Run Module (or press the keyboard shortcut F5). Python will read each line of your program and execute the commands. The commands print a line of * characters, the welcome message, and another line of * characters to the screen (see figure 3.8).

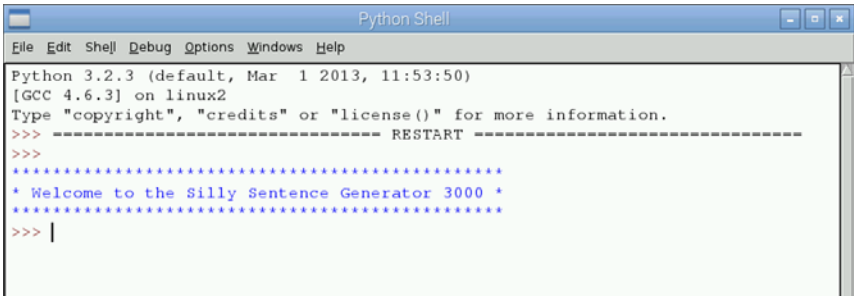


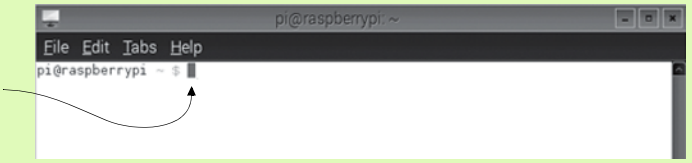
Figure 3.8 Running the program SillySentence.py displays a welcome message on the screen.

Excellent! Now you have a proper welcome message for your game. The next thing you need to do is gather some input from your game player. Some games use button presses, but you'll use the keyboard for this game.

Running programs from the command line

Another way to run a program is from the Raspbian command line. You can access the command line using the Terminal application found under Menu-->Accessories. A window will open with this prompt:

```
pi@raspberrypi ~ $
```



The terminal shows a prompt, ready for your commands.

To run the Silly Sentence program at the command line, enter

```
pi@raspberrypi ~ $ python3 SillySentence.py
```

The next figure shows this command and the result. Notice that you get the same output at the command line.

The command line is another option for running Python programs. In part 3 of this book, you'll see that some programs require you to run them from the command line because you must run them as the superuser on your Raspberry Pi.

Adding notes in your code

Imagine a comic book without words. You'd have a hard time understanding what was happening from just the pictures. Maybe you could figure it out if you studied the comic long enough, but words are important for understanding a story. Lines of code can be like a comic book without words: you know something is happening, but you might not be able to tell what without guessing.

That's why programmers invented the idea of adding comments. *Comments* are notes in the code that explain what's happening. They're as much for you as for other people who may read your code. You can use comments to explain why you wrote the program and how parts of the program work.

Using hashtags for comments

You add a comment by starting the line with a hashtag (#) and a space and then typing in your comment text. Let's add comments to the beginning of Silly Sentence Generator 3000 to explain the program's title, its purpose, and who wrote it.

Listing 3.1 Adding notes to your program

```
# Title: The Silly Sentence Generator 3000
# Author: Ryan Heitz
# This is an interactive game that creates funny sentences
# based on input from the user

# Display a welcome message
print("*" * 48)
print("* Welcome to the Silly Sentence Generator 3000 *")
print("*" * 48)
```

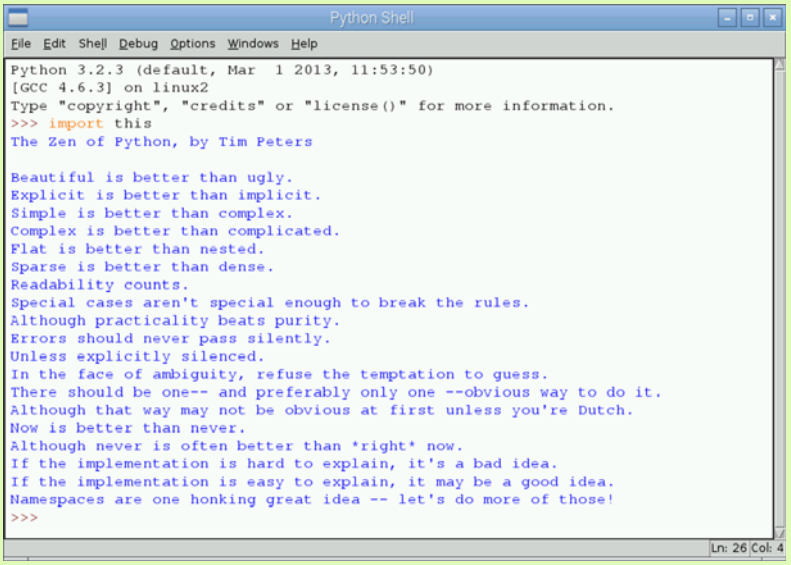
Lines beginning with hashtags are comments and are ignored by Python.

Displays a welcome message

Comments are helpful to the humans reading the code. But Python ignores comments when it runs your program. You can check this by saving your program and running it again; you'll see that you get the same result as before.

Easter egg: the Zen of Python

Python has a hidden surprise regarding Python style. In computer programs, these surprises are sometimes called *Easter eggs*. You can find the egg by typing `import this` in the Python Shell and pressing Enter. A beautiful poem called "The Zen of Python" will appear on your screen.



The poem emphasizes the philosophy of Python. Some of it talks about advanced topics, but many lines discuss a way of coding that is meant for anyone who uses Python. The seventh line captures a great idea in Python: "Readability counts." It's better to write programs using simple instructions that are easy to read than to try to mash together steps in complicated, long lines of code. Try taking some deep meditational Python breaths before getting back to your project.

Python's creator, Guido van Rossum, said that code is read more often than it's written.[2] Readability is an extremely important part of programming and is a guiding principle in the style of Python programs. Comments are an important way to keep your code easy to read and understand.

Comments are your new friend, and they will make your code easy to read. You'll keep using them to add notes to your code as you collect information from your game player (or user) and create a silly sentence.

Getting and storing information

To gather input from users, you can use the `input` function. Let's add a line of code in your program that will ask the user for their information and store that information in a variable.

Listing 3.2 Gathering input from the player

```
# Title: The Silly Sentence Generator 3000
# Author: Ryan Heitz
# This is an interactive game that creates funny sentences
# based on input from the user

# Display a welcome message
print("*" * 48)
print("* Welcome to the Silly Sentence Generator 3000 *")
print("*" * 48)

# Get the user's name and say hi
player_name = input("Please enter your name: ")
```

Gathers input from the user

[2] Check out the resource *PEP 8—the Style Guide for Python*, written by Python's creators: http://legacy.python.org/dev/peps/pep-0008. A wonderful section called "A Foolish Consistency Is the Hobgoblin of Little Minds" talks about the importance of readable code.

When you use the `input` function, it displays a prompt and awaits the user's reply. After the user enters something and presses Enter, the information is stored in the variable on the left side of the equals sign.

In the IDLE editor, `input` shows up in purple highlighting, indicating that it's the name of a function in Python. Let's look closely at the `input` function to see how it works (see figure 3.9).

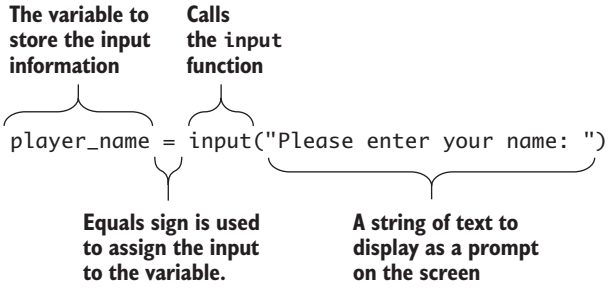


Figure 3.9 The `input` function displays a prompt to the user. The prompt "Please enter your name: " tells the user what you want them to type in or enter. In this case, you're asking for their name.

On the right side of the equals sign, the `input` function is called, and you open a set of parentheses. You can give the `input` function a string that acts as the prompt. This is the message that is displayed on the screen and that says to the user, "Hey you, please type something in"—only more nicely! Make sure your string starts and ends with quotation marks (`""`).

Run the program by pressing F5 or selecting Run > Run Module. The program displays the welcome message and then an input prompt with a blinking cursor. Python is waiting for your input: it needs you to type something in and press Enter.

On the left side of the equals sign is the name of a variable in which the information will be stored. When you type something in and press Enter, the value of what you typed is stored in the variable `player_name` as a string.

> PYTHON 2.X The `input` function was previously `raw_input` in Python 2.X.

Joining strings

As in other apps and websites, you want the user to feel welcome, so let's use their name and give them a proper greeting. A nice message to display on the screen might be

```
"Hello, Ryan! Let's make a silly sentence!"
```

To create a personal feel, you'll create just such a message that joins the user's name with some words welcoming them. You use the plus (+) symbol to join strings:

```
message =  "Hello, " + player_name + "!  Let's make a silly sentence!"
```

If `player_name` equals "Melissa", the message is equal to

```
"Hello, Melissa!  Let's make a silly sentence!"
```

Add this to your program, and display the message to the screen using `print`.

Listing 3.3 Using + to join strings

```
# Title: The Silly Sentence Generator 3000
# Author: Ryan Heitz
# This is an interactive game that creates funny sentences
# based on input from the user

# Display a welcome message
print("*" * 48)
print("* Welcome to the Silly Sentence Generator 3000 *")
print("*" * 48)

# Get the user's name and say hi
player_name = input("Please enter your name: ")
message = "Hello, " + player_name + "!  Let's make a silly sentence!"
print(message)
```

Joins strings

Displays the message

The program has the user input their name, which is stored in the variable `player_name`. On the next line, a message is made by joining strings. The message is displayed on the screen to create a personalized start for the game.

More tools for strings: string methods

To make life easier, Python includes some built-in tools for working with strings. These tools are similar to the functions you saw earlier, but they're called *methods*. Here is an example of a method that capitalizes the first letter of a string:

```
"jOHn".capitalize()
```

The `capitalize` method converts "jOHn" to "John".

Python has a whole set of built-in methods. One method for strings is the `lower` method, which converts a string to all lowercase:

```
"RABBIT".lower()
```

This makes "RABBIT" turn into "rabbit".

Another method, `upper`, makes all the letters uppercase:

```
"king Arthur".upper()
```

The `upper` method is great for shouting things. It makes "king Arthur" into "KING ARTHUR".

These methods can save you time[a] and make it easier for you to get things done.

Methods vs. functions

Methods are a type of function, but they use *dot notation*. This means you put a period (`.`) after the item and then the name of the method. If your item was "John Cleese" and the method you wanted to use was `lower`, you'd write

```
"John Cleese".lower()
```

Parentheses go after the method name. You put in the parentheses any inputs required by the method. You can check the Python documentation online to see what is required.

Some methods don't require any inputs, like the string methods `capitalize`, `upper`, and `lower`. But some methods, like `count`, require inputs. Imagine that you had a set of test answers with `T` for true and `F` for false, and you wanted to count the number of true answers. You could use `count`:

```
>>> TestAnswers = "TTTFFFTTTFTFFFFTTTFFTT"
>>> TestAnswers.count("T")
12
```

There were 12 true answers on the test.

[a] You can learn more about the available string methods in the online Python documentation: http://mng.bz/9z49.

Let's go further and add more inputs.

Using more than one input

You have a wonderful start to your game. Now you need to gather multiple inputs from the player. Let's start by asking the player for a noun—the name of a famous person:

```
famous_person = input("Enter the name of a famous person: ")
```

Next, you should get a few more words:

```
adjective1 = input("Enter an adjective: ")
adjective2 = input("Enter another adjective: ")
verb = input("Enter a verb ending in -ING: ")
```

With these multiple inputs, your code should now look like the following listing.

Listing 3.4 Collecting multiple items from the player

```
# Title: The Silly Sentence Generator 3000
# Author: Ryan Heitz
# This is an interactive game that creates funny sentences
# based on input from the user

# Display a welcome message
print("*" * 48)
print("* Welcome to the Silly Sentence Generator 3000 *")
print("*" * 48)

# Get the user's name and say hi
player_name = input("Please enter your name: ")
message = "Hello, " + player_name + "!  Let's make a silly sentence!"
print(message)

# Gather words from the player for our sentences
famous_person = input("Enter the name of a famous person: ")
adjective1 = input("Enter an adjective: ")
adjective2 = input("Enter another adjective: ")
verb = input("Enter a verb ending in -ING: ")
```

Gather words from the player.

You use the `input` function multiple times to collect a set of words from the user. Each word is stored in a variable on the left side of the equals sign. Try to use names for variables that make sense; it'll be easier to remember what you stored in them later.

Building the sentence

Now let's create the sentence for your Silly Sentence Generator 3000 by joining the words using +:

```
silly_sentence = ("The " + adjective1 + " " + player_name + " is " +
                  verb + " the " + adjective2 + " " + famous_person)
```

Let's take a closer look at this line of code in figure 3.10 to see what's happening.

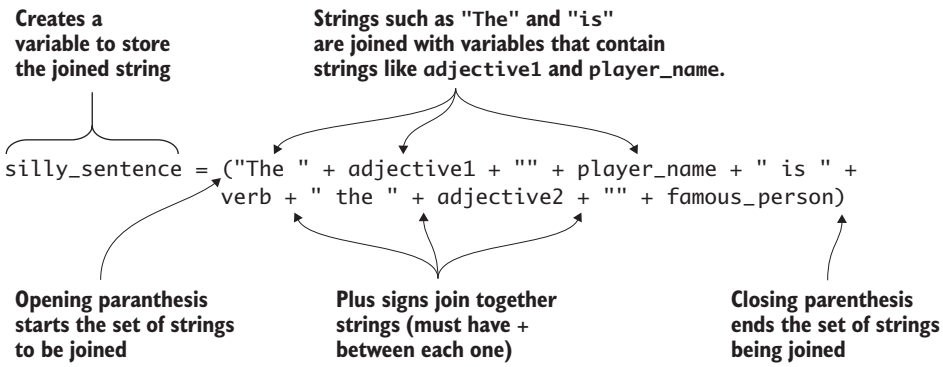


Figure 3.10 `silly_sentence` is created by joining a set of strings. The strings are a combination of strings you enter with quotation marks around them and strings collected from the game player that are stored in variables. The parentheses are needed because the code is too long to fit on a single line.

On the right side of the equals sign, the parentheses enclose the strings that are being joined to create a sentence. They're joined (or *concatenated*) using the + operator. Because the line is so long, you can use a set of parentheses to break it over two lines. Python recommends limiting all lines to no longer than 79 characters so the code can be easily read. Looking at the left side of the equals sign, you'll see that the resulting string is stored in a variable named `silly_sentence`.

What's especially awesome is that this code will create a different sentence each time a user enters different words. Because you used

variables and the variables are storing the input from the user, it's truly a Silly Sentence Generator!

Troubleshooting

When typing code, it's easy to make mistakes, called *bugs*. Boo to bugs. To track them down and fix them, you *debug* your code. Yay for debugging. You may forget to close a set of quotation marks, you may leave out a parenthesis, or you may misspell a word. Let's look at some common errors you might make and how to fix them.

In the last section, you used the + to join strings and variables that were storing strings. Look at this code, which has an error:

```
silly_sentence = ("The  + adjective1 + " " + player_name + " is " +
                  verb + " the " + adjective2 + " " + famous_person)
```

Do you see the problem? The first string ("The) is missing the closing quotation mark ("The "). If you were to run this program, Python would output an error (see figure 3.11). Add the closing quotation

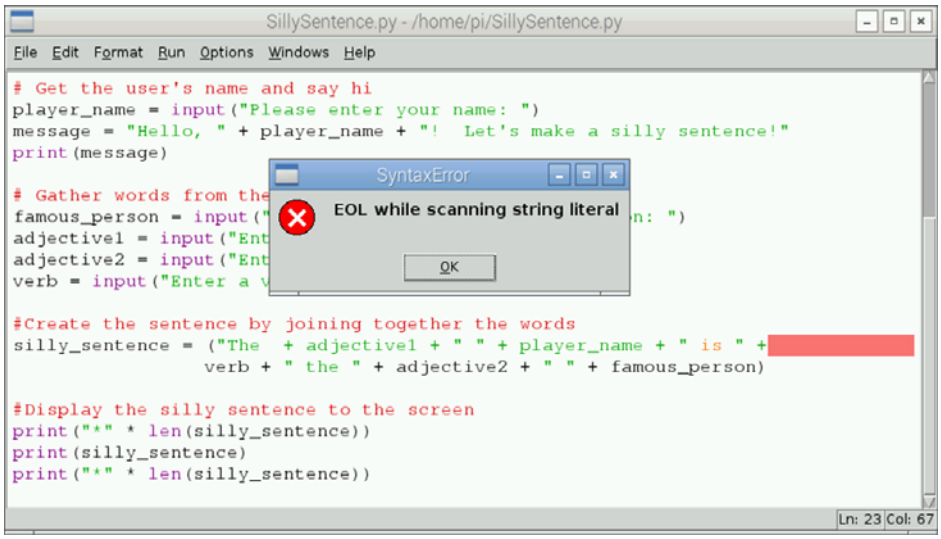


Figure 3.11 If you forget to close a set of quotation marks around a string, you'll receive an error from Python when you try to run your program. Python will highlight in red the line with the error. Check each of the strings to find and fix the error.

mark to the string that is missing it, and then save your program and run it again.

Another common error you might make is to misspell the name of a variable or use different capitalization. Here's the same line of code, but this time there is a misspelled variable and one variable with incorrect capitalization. Can you spot them?

```
silly_sentence = ("The " + adjectve1 + " " + player_name + " is " +
            verb + " the " + adjective2 + " " + Famous_person)
```

The first one is `adjectve1`, which should be `adjective1` (the `i` is missing). The second error is `Famous_person`, which should be `famous_person` (the `F` should be lowercase). The error you'll see if you run this program is shown in figure 3.12.

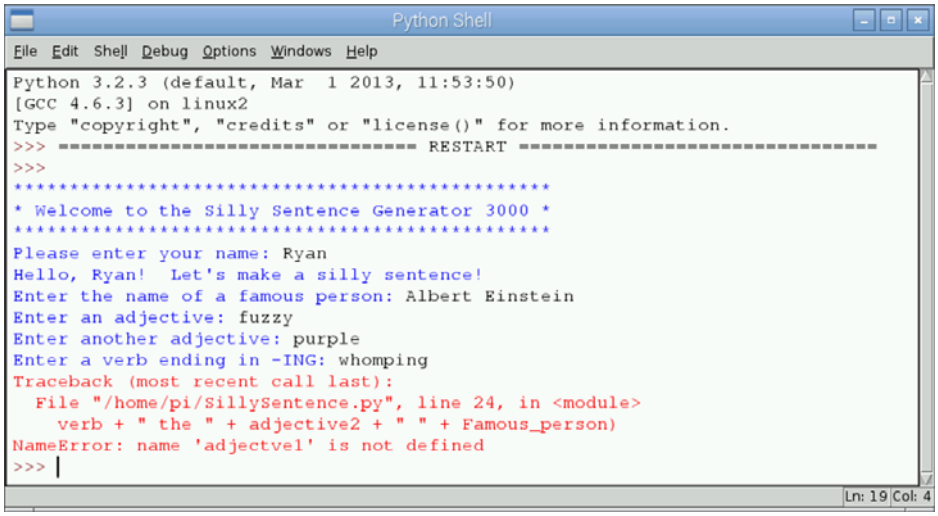


Figure 3.12 A common mistake in programming is to misspell the name of a variable or use incorrect capitalization. The error displayed says there is a problem on line 25 of the program. The type of error is `NameError: name 'adjectve1 ' is not defined`.

> TIP The spelling and capitalization of a variable must always be the same. If you call a variable `my_number` and then later type `my_nomber` or `My_number`, Python will give you an error.

Correct the error by fixing the spelling of `adjectve1` so it's `adjective1`. After fixing it, you'll still receive an error, but this time because of the capitalization of `Famous_person` (`NameError: name 'Famous_person' is not defined`). Change the capitalization of `Famous_person` to `famous_person`. Once you've made the corrections, save the program and run it again.

You've debugged your program. Superb job!

Completing the program: displaying the silly sentence

You've made your silly sentence, and you want Python to show it to the player. Use the `print` statement to print it out, but like your welcome message, let's add some pizzazz to it!

```
print("*" * 48)
print(silly_sentence)
print("*" * 48)
```

Guess what it does? It prints a row of * characters (asterisks) across the screen 48 times. Then it displays the sentence and prints another row of * symbols 48 times. Try other characters or patterns of characters to see what looks good to you!

It looks pretty good, but you can do a bit better. Test your program by running it, and you'll notice the number of symbols doesn't match the length of the sentence. You've programmed it to display exactly 48 asterisks—no more, no less. Instead, let's update those lines to repeat the symbol to match the length of the silly sentence. You'll use another built-in Python function called `len`, which calculates the length of a string and returns a number telling you the number of characters:

```
print("*" * len(silly_sentence))
print(silly_sentence)
print("*" * len(silly_sentence))
```

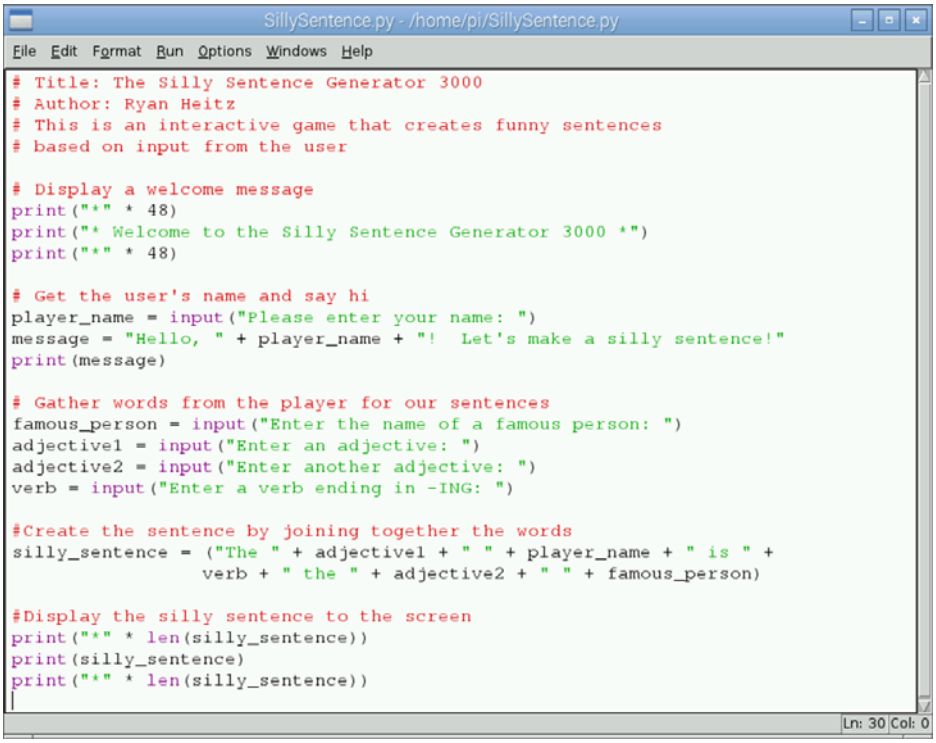


Figure 3.13 Silly Sentence Generator 3000 is a fun program that shows how programs can collect information from users, interact with them, and provide a more personal feel.

That's better! Let's look at the code all together (see figure 3.13).

You've completed your program. Let's do some final testing to see what it can do! See figure 3.14 for an example of the game's output.

Fantastic! Feel free to update the code to add more adjectives, verbs, or nouns. You've learned how to get input from a computer user and interact with them by displaying a message to the screen.

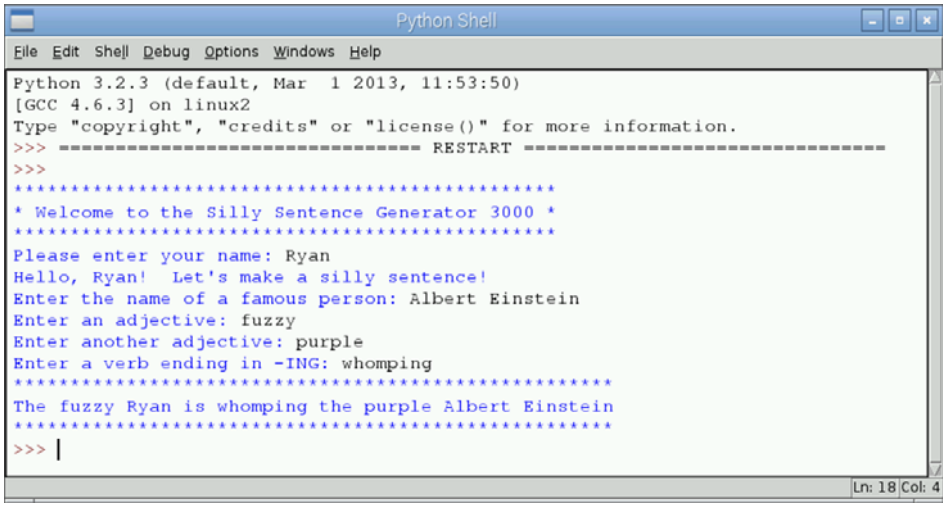


Figure 3.14 The Silly Sentence Generator 3000 makes some absurd sentences based on words you enter.

Fruit Picker Extra: Minecraft Pi

In this Fruit Picker Extra, you'll explore another unique feature of the Pi: it has its own version of Minecraft. Thanks to a collaboration between Mojang, the makers of Minecraft, and the Raspberry Pi Foundation, a free, slimmed-down version of Minecraft is available on the Raspberry Pi. Since September 2014, this version, called Minecraft Pi, is automatically installed with the Raspbian operating system.

What's Minecraft?

Minecraft is a game that takes place in a 3D virtual world made of blocks. At the most basic level, you run around mining (digging blocks by hitting them) and crafting things (combining items in the game to make new items). You can also build things in this virtual world using different types of blocks.

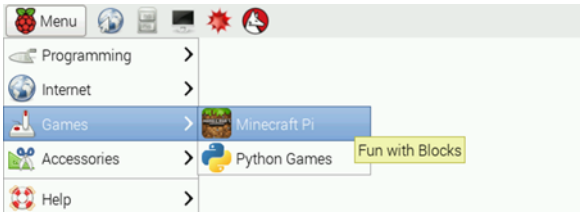


Figure 3.15 Minecraft Pi is a slimmed-down, free version of Minecraft that's based on Minecraft Pocket Edition. It's limited compared to the full version but still oodles of fun!

Launching Minecraft Pi

Look for a Minecraft Pi icon under Menu > Games (see figure 3.15). If you got your Pi before September 2014, see the chapter 6 sidebar "Updating your Pi" to learn how to update Raspbian.

Click the Minecraft Pi icon to open the game. A Minecraft window will open (see figure 3.16). It's a little quirky—you'll see a black window behind the Minecraft window—but this is normal.

Click Start Game to begin to play. Next, click Create New to create a new world. After it's done loading, you'll find yourself in a blocky

Figure 3.16 The Minecraft Pi main screen allows you to start a single-player game or join a multiplayer game. The multiplayer option lets you connect to someone else's world, but you'll need to be on the same network.

Figure 3.17 Each Minecraft world is made of blocks but is different. You might find yourself in a forest or in a desert. The bottom of the screen shows you the items in your inventory. Use the mouse scroll wheel to select different items, or press the numbers 1–9 on your keyboard.

world (see figure 3.17). Each world is different, so you may see trees, water, dirt, or any number of environments.

In Minecraft, you're a player who can walk around using the following controls:

- *W*—Move forward.
- *A*—Move left.
- *S*—Move backward.
- *D*—Move right.
- *Spacebar*—Jump.
- *Mouse movement*—Look around or turn.
- *Escape*—Exit the game.

In addition to the basics, here are some other moves you may need:

- *Double spacebar*—Fly up in the air (double-tap the space bar and then hold it down to fly up). Press the left Shift key to move down. If you're flying, double-tap the spacebar to fall back to the ground.
- *E*—Show the game inventory of blocks and items you can use (it's limited compared to the full version of the game). Drag items you

want to the small squares at the bottom of the screen. Press Escape to hide the inventory screen.

- *Scroll wheel or the number 1–9 keys*—Select something from one of your player inventory spots at the bottom of the screen. The item selected is in your hand for you to use.

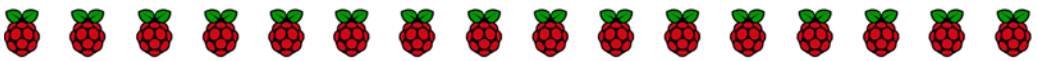
Once you get the hang of moving around, use the mouse left click to dig or break blocks. Use the mouse right click to place a block or use the tool in your hand. When you're ready to leave, press Escape to exit the game.

> TIP To exit Minecraft Pi, press Escape > Quit to Title, and then click the X in the corner to close the window.

Python programming interface to Minecraft Pi

Minecraft Pi has a fun inventory of materials and tools—even a sword! What's even better is that there is a Python programming interface for Minecraft Pi. Head over to the Raspberry Pi Foundation website to learn more about how to use Python to interact with Minecraft Pi.

Explore the world, dig an underground base, or build a tree house. What will you do?



Challenges

Try these challenges to see if you can use the `input` function and strings to create something fun and interactive.

Knight's Tale Creator 3000

In this challenge, try to use what you've learned about input (gathering text) and output (displaying text) to create a Knight's Tale Generator. Here is a story template for you to use:

> *There was a brave knight, [player_name], who was sent on a quest to vanquish the [adjective] evildoer, [famous_person]. Riding on his/her trusty [animal], the brave [player_name] traveled to the faraway land*

of [vacation_place]. [player_name] battled valiantly against [famous_person]'s army using his [sharp_thing] until he defeated them. Emerging victorious, [player_name] exclaimed, "[exclamation]!!!" I claim the land of [vacation_place] in the name of Python.

The words in brackets are meant to be variables that you'll create in your program; you'll need to have the player input those words. Remember to use + to join the strings to create a unique knight's tale, and then print the tale to the screen. Good luck!

Subliminal messages

A *subliminal message* is a hidden message that tries to get people to think of something you want them to think about. Often used in TV commercials, it's a great technique to try with friends and parents to get something you want.[3] In this challenge, try to create a message that is hidden in a large display of characters. The message should be

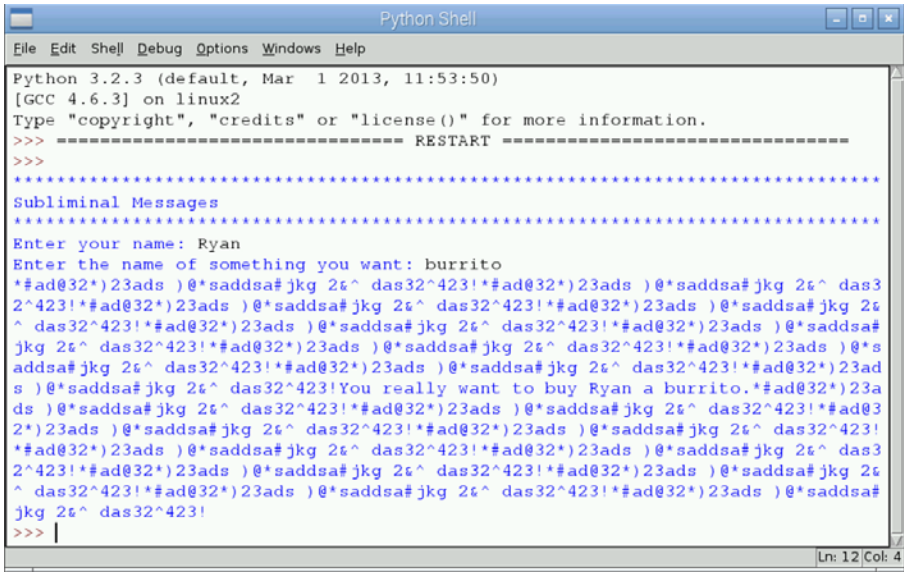


Figure 3.18 The subliminal-message challenge is about hiding a secret message in a bunch of characters on the screen. Can you see the hidden message?

[3] Use subliminal messaging at your own risk (send Ryan pizza!). If people know you're trying to manipulate their minds, they may retaliate with subliminal messaging of their own.

constructed by asking for the person's name, the name of something they want, and a pattern of letters, numbers, and symbols. In your program, you should create a message that says, "You really want to buy [player_name] a [thing]", and hide it within a pattern of characters. Figure 3.18 shows an example.

In this example, the hidden message is, "You really want to buy Ryan a burrito." Be sneaky, and see if you can find a way to create and hide a subliminal message!

Summary

In this chapter, you learned how to write interactive programs that get information from a person and provide entertaining responses:

- Use the `input` function to collect text input from a person. Use it with a variable and an equals sign to store the information that a person types in. Here's an example of asking the user to tell you their favorite color and saving it to a variable called `favorite_color`:

```
favorite_color = input("What is your favorite color?")
```

- Add notes to your programs by starting a line with a hashtag (`#`) and a space:

```
# A comment tells you about the code
# They help you read the code,
# but they are ignored by Python
```

- Join strings using `+`.
- Use parentheses when you need to join strings that are longer than a single line:

```
name = input("What is your name?")
favorite_color = input("What is your favorite color?")
message = ("Your name is: " + name + " and your "+
           "favorite color is: " + favorite_color)
```

The game you created uses the same ideas to collect information from users and interact with them in the same way they see every day on websites, mobile apps, and games.

4

Norwegian Blue parrot game: adding logic to programs

In this chapter, you'll learn how to create Python programs that

- *Display an introduction*
- *Collect input from the user*
- *Use `if` statements to respond to users in different ways*
- *Use `while` loops to repeat things over and over*
- *Use Python code libraries to generate random numbers*

Open a popular game, such as Minecraft, or think about a robot, like the Mars rover. Both are computer programs. What do they have in common? They both have the ability to take input and do something with it. What they do depends on the input they're given. In a game, if you press Forward and fall off a ledge, your character falls and dies. If it's your only life, then you're taken to the Game Over screen. Similarly, the Mars rover might be instructed to go to a certain location, but if it detects a large rock in its way, it will stop or attempt to drive around the obstacle.

The logic of how games work or how the rover moves is programmed into them. But how do you create that logic in your programs? You'll learn

how by making a simple guessing game about a special parrot, the Norwegian Blue.

Displaying the game introduction

The Norwegian Blue parrot is a fictitious parrot that is the subject of one of the most famous comedy sketches from Monty Python.[1] Your game is about pretending you're visiting a pet shop that has a Norwegian Blue parrot for sale. The shop owner challenges you to guess the age of the parrot (see figure 4.1). If you guess correctly, then you get to take home the parrot for free.

Figure 4.1 The Norwegian Blue parrot has beautiful plumage and makes a great subject for a guessing game.

Each time the game is played, the program selects a different random number between 1 and 20 as the age of the parrot. The game player gets five chances to guess the parrot's age. If the player guesses correctly, the game displays a funny message congratulating them on winning their new parrot. If the player makes a wrong guess, then the program

[1] If you haven't seen it, check out this Wikipedia page, which has an audio recording of the comedy sketch: http://en.wikipedia.org/wiki/Dead_Parrot_sketch.

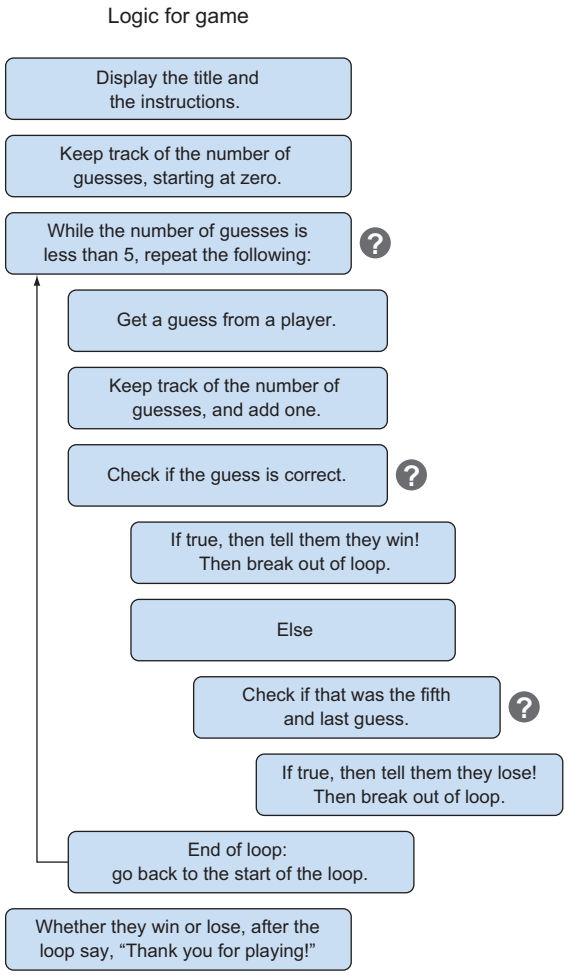


Figure 4.2 The game logic can be expressed in words. The question marks symbolize when the game needs logic to make a decision. This diagram also shows what code needs to be repeated because the player gets five guesses. Each decision has a simple True/False or Yes/No answer.

displays a good-hearted insult, as if it were offended by the player's guess. If the player doesn't guess within five tries, they lose, and the pet shop owner lets them know the parrot's true age (see figure 4.2).

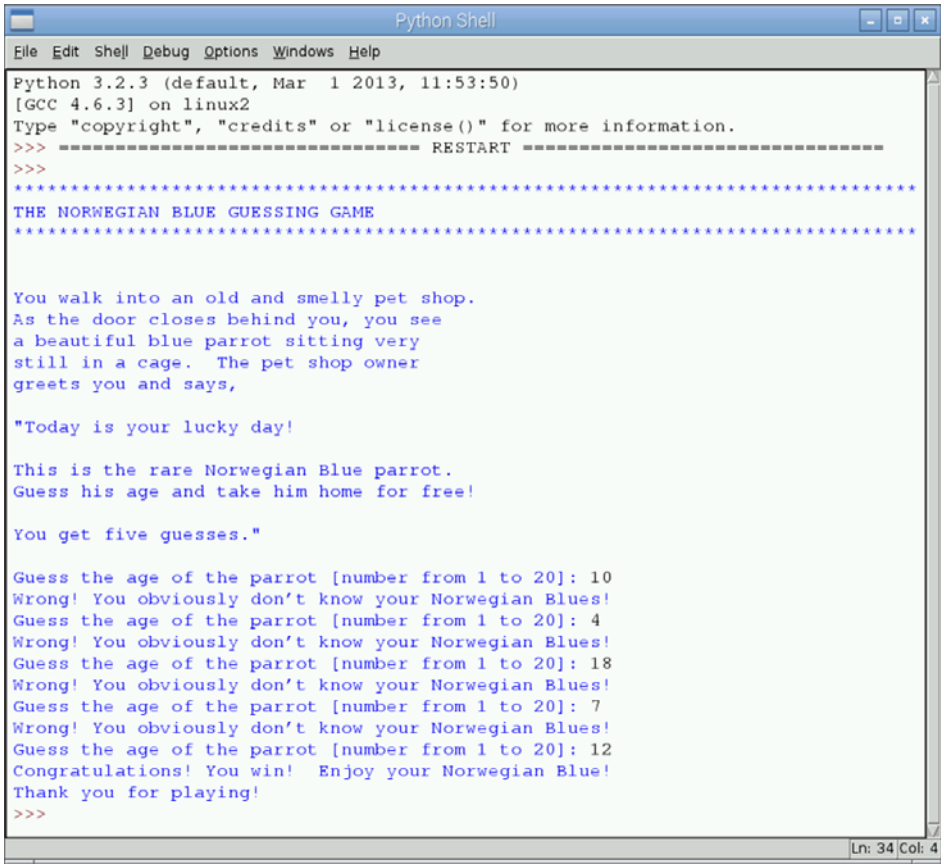


Figure 4.3 The Norwegian Blue Guessing Game is about trying to guess the age of a bird in a pet shop.

When this game is completed, you'll be able to play it. The output will look like figure 4.3. In the example, the player guessed four times incorrectly; but on their fifth try, they guessed correctly. They won, and the shop owner gave them the parrot.

Creating the game welcome message and instructions

Let's start by opening IDLE for Python 3 and creating a new program. Open IDLE by clicking the Python 3 icon under Menu > Programming on your Raspberry Pi desktop (see figure 4.4).

Figure 4.4 Click the Python 3 icon to open IDLE's Python 3 Shell on your Raspberry Pi.

Give your Raspberry Pi a few seconds to open IDLE. You'll see the Python Shell. Press Ctrl-N or File > New Window to open the IDLE text editor. You'll see a blank window, ready for you to start typing in the program.

Let's type in a few comments at the top of the program in the text editor. Start each line with a hash tag (#) and a space.

Listing 4.1 Creating comments at the top of your new program

```
# Title: The Norwegian Blue Parrot Guessing Game
# Author: Ryan Heitz
# The goal of the game is guess the age of a parrot.
# The program generates a random age between 1 and 20.
# The player gets 5 guesses to guess the age correctly.
# If they're correct, they win the parrot!
```

Change the words to whatever you'd like. Comments are notes for you and whoever you might share your program with, so make them read the way you want. Remember to avoid going off the screen with your comments—keep each line pretty short. No more than 79 characters per line is good style; this ensures that your beautiful Python programs fit in the window and don't require the user to scroll or resize the window.

> TIP You can keep track of which line and column your cursor is on by using the cursor-location information (see figure 4.5). It's displayed in the bottom-right corner of the text editor. The letters *Col* stand for column: this shows the number of characters your cursor is from the left side of the screen. The left side is 0, the middle is 40, the right side is 80, and so forth.

The program in the IDLE text editor now contains several lines of comments. Before you go further, save your work: press Ctrl-S to save the program. A window will pop up in which you can name and choose a location in which to save the file. In the File Name text box, type in the name of the file: name it `NorwegianBlue`. When you click the Save button, the file will be saved as NorwegianBlue.py (the .py extension is automatically added by IDLE), and it will be stored on your Pi's SD memory card in the /home/pi folder.[2] Once the program is saved, the text editor displays the location of the file and the filename along the top of the window (see figure 4.5).

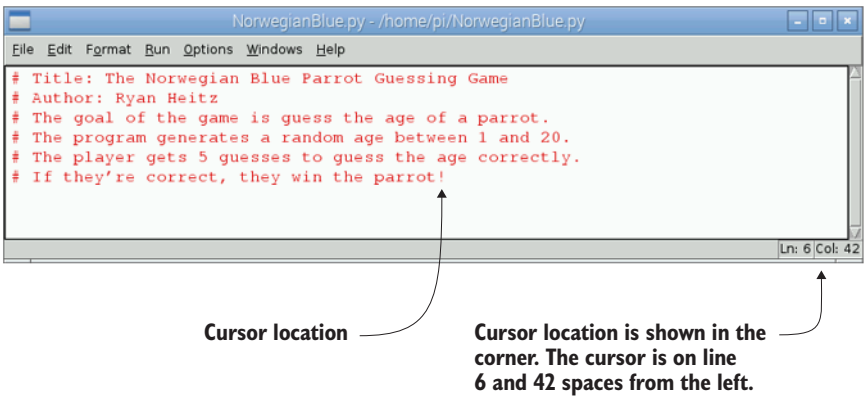


Figure 4.5 Once you've saved the file, the top of the window displays the filename and the location where the file is stored on your Raspberry Pi (/home/pi/NorwegianBlue.py). The cursor location is always shown at the bottom right of the window.

[2] You can create a new folder in which to store your Python program. You create a folder by opening the Raspbian File Manager application and selecting File > Create New > Folder. Like your shoes, you'll want to remember where you stored your programs so you don't have to spend a lot of time looking for them.

Next you need to let the user know the name of your game and the instructions for playing it. Use Python's built-in `print` function to write a few lines of code that display a title on the screen.

Listing 4.2 Making the title display on the screen

```
# Display the title and instructions
print("*" * 80)
print("THE NORWEGIAN BLUE GUESSING GAME")
print("*" * 80)
```

Draws a line of asterisks on the screen

Add a comment

Displays the title of the game

After they see the title, your game players need to know what to do. You should set the scene for the game and give them instructions. Let's create a variable called `instructions` and store in it the sentences describing how to play the game. As in Silly Sentence Generator 3000 from chapter 3, this variable will contain a string of characters a few sentences long.

Rather than enter a super-long string all on one line, you want to use a neater way to keep the string on the screen and limit it to not more than 79 characters across (remember, good Python style is to keep text on the screen). In Python, you can use string literals to do this.

> DEFINITION *String literals* are strings that can hold multiple lines of text and that appear exactly as you typed them in the text editor when they're displayed on the screen. String literals keep the spaces between lines and characters. To make one, start and end a string with triple double quotes (""") or triple single quotes (''').

Let's add instructions to your program after the program's comments. You'll use a string literal for the instructions and then print it to the display.

Listing 4.3 String literals that hold multiple lines of text

```
instructions = """
You walk into an old and smelly pet shop.
As the door closes behind you, you see
a beautiful blue parrot sitting very
still in a cage. The pet shop owner
```

Start a string literal with """ or '''.

```
greets you and says,
"Today is your lucky day!
This is the rare Norwegian Blue parrot.
Guess his age and take him home for free!

You get five guesses."
"""
print(instructions)
```

You can use double quotes in string literals.

End the string literal with """ or '''.

Displays the instructions in the Python Shell window.

String literals give you the ability to display a string exactly as you type it in the text editor. Think of it as a "what you see is what you get" way of creating strings.

Getting expressive with ASCII art

Before desktop operating systems (OSs) and games had high-end graphics, computers had limited display capabilities. Computer users and programmers invented a new type of art called *ASCII art* that uses text characters to make images.

ASCII is a way of storing characters as binary numbers. For example, the letter *A* is represented as 1000001. Later encodings had many more characters to support more languages, but the name *ASCII art* stuck. ASCII art uses the set of 95 ASCII characters (letters, numbers, and symbols) in cleverly designed patterns to represent images.

Here is an example of ASCII art for your game title that is made by creating a string literal and printing it to the screen. Craft your own ASCII art using a bit of imagination and trial and error:

```
bird_art = """
```

```
########################################################################
                        ____
                       /  0 \          NORWEGIAN
                      |      >
                      |UUU) |          BLUE
                      |UUU) |
                     //UUU) |          GUESSING
                    //UUU)  /
                   //UU)   /           GAME
                  //U)    /
                 // -|--|/
               ==// ==W==W====
                //
               /
########################################################################
"""

print(bird_art)
```

Sometimes it helps to blur your eyes a bit to see if the image looks like what you want. Get creative, and think how you can use uppercase and lowercase letters to create effects, like using a *U* to represent feathers on the parrot's wing or *W* for the parrot's claws.

Try these ASCII art sites for fun:

- *www.chris.com/ascii*—A huge collection of ASCII art, sorted by topics
- *http://patorjk.com/software/taag*—A text-to-ASCII art generator (TAAG). You type in words, and it automatically creates ASCII art for you.
- *http://picascii.com*—A tool that converts pictures to ASCII art

See if you can make some ASCII art for the title screen of your game that's even better than this. Have fun with it!

It's always a good idea to test your programs often to catch any mistakes. Test your program now, and see what you get. The title and instructions should display nicely on the screen.

A common mistake you might make when typing in this code would be to forget some of the quotation marks at the beginning or end of the

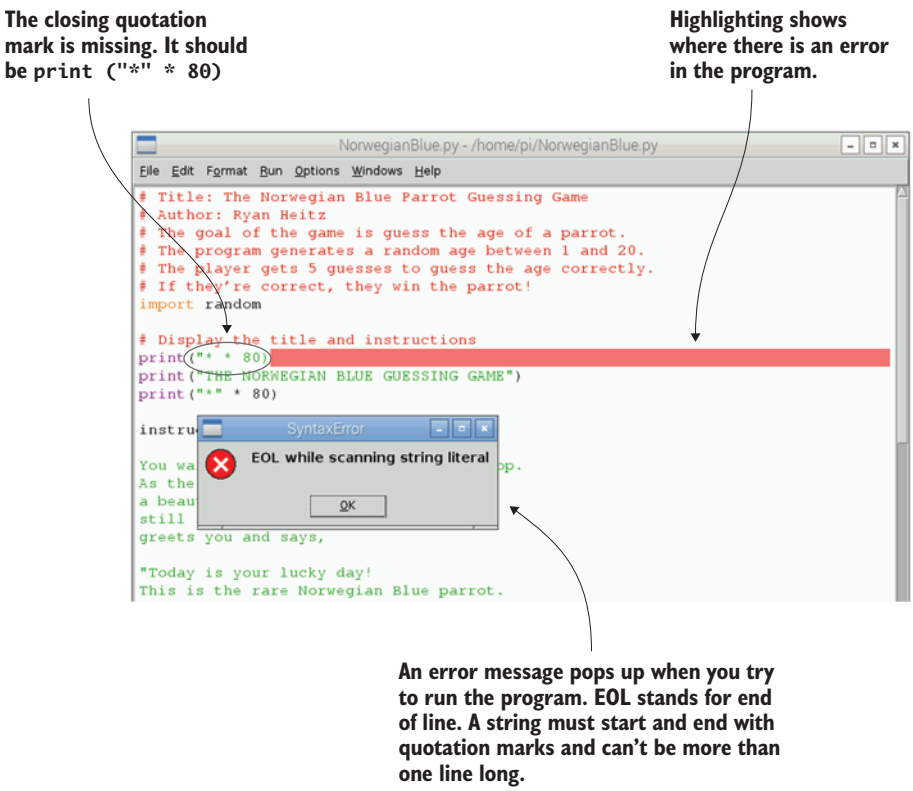


Figure 4.6 Python will display an error if you forget starting or ending quotation marks. The line with the error will be highlighted in your program. Fix the program by adding the missing quotation marks, and then save and run the program.

strings. If you do, figure 4.6 shows an example of the error you'll receive in Python.

A similar mistake you might make is forgetting to start your string literals with a triple quotation mark. In this case, Python will give you a syntax error message (see figure 4.7).

It's easy to fix this error by making sure there are triple quotation marks at the beginning and end of the string literal. Use the highlighting shown in the IDLE text editor to figure out which line is causing the problem.

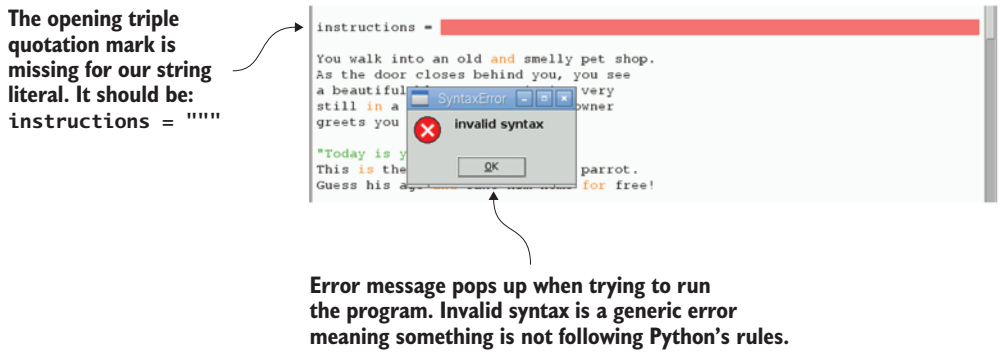


Figure 4.7 A string literal must start and end with a set of triple quotation marks. If you forget, Python will tell you that you have a syntax error. Add the missing triple quotes to fix the error.

Collecting input from the player

Your game has a proper introduction; now let's start interacting with the player. Games, websites, and apps are all about causing interactions, whether it's to create some fun or help you buy something online. Contrast that with the last movie you watched. Movies don't have any interaction—they're always the same.

A computer program's ability to accept input and respond to that input is special. In text-based games like the one you're creating, this interaction occurs through the keyboard. Players type in answers or make choices, and the game responds.

For this game, you want to ask the game player to guess the age of the parrot. The program knows the parrot's age and checks whether each of the player's guesses matches it. To make this work, you have to give the program the age of the parrot (it's stored in a variable). This gives you something akin to god-like powers as the programmer—as the game's maker, you can decide what the value is. Let's create a variable and set it to a value that you pick. One great thing about being a computer programmer is that only you know the parrot's true age.[3]

[3] And anyone else who is reading this book! Later you'll make the game use a random number so even you don't know the parrot's age.

Let's make the parrot old. Create a variable named `parrot_age`, and assign it a value of 19.

Listing 4.4 Creating an age for the parrot

```
# Making up the parrot's age
# TODO: Make this automatically pick a random number between 1 and 20
parrot_age = 19
```

Create a variable and store the number 19 in it.

Notice that in the comments you include a `TODO` note: this tells you that you have an item to do later.

> **TIP** Use `TODO`s in your comments as reminders of areas of your program that are left unfinished or need further improvement. Comments are your friend, and they're there to help you. Use them however you need them!

Next let's get the user's first guess. Use Python's `input` function (like you did in chapter 2) to collect input from the user and store it in a variable named `guess`. Give the `input` function a message that clearly prompts the game player to enter an appropriate value. You don't want them typing in 50 when you're expecting a number between 1 and 20. In this case, you want them to guess a number from 1 to 20.

Listing 4.5 Getting a guess and storing it in a variable

```
# Get a guess from the user
# TODO: Need to make this repeat to give them five guesses
guess =input("Guess the age of the parrot
➥ [Pick a number from 1 to 20]: ")
guess = int(guess)
```

Leave a note to yourself to make this program allow the user to guess five times.

Display a prompt to the user and store whatever they type into a variable called guess.

Change the value stored in `guess` from a string to an integer.

After gathering input from the user, you need to convert the value from a string (for example, "5") into an integer (simply the integer 5). By default, anything input by the game player is stored as a string (even if what they type in is a number). Figure 4.8 shows this graphically:

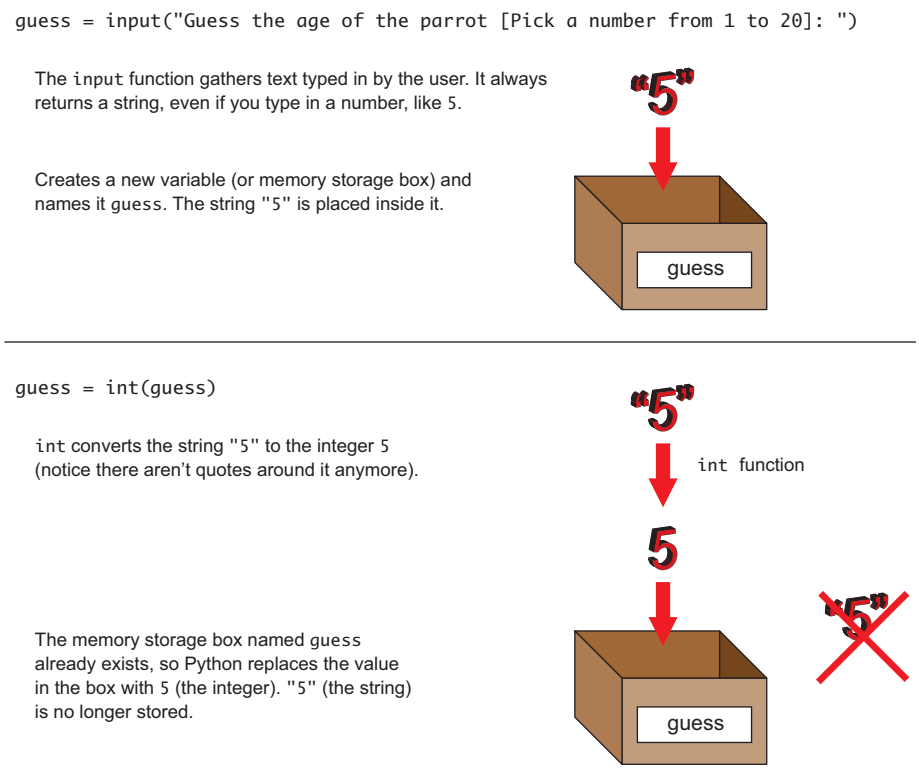


Figure 4.8 The `input` function gathers text typed in by the user; then the text is stored in a variable as a string data type. You take the value of the variable ("5"), convert it to an integer (5), and store it as the variable.

you're gathering input from the user and then converting it to an integer. The `int` function takes the value in the `guess` variable, converts it to an integer, and then stores it back in the `guess` variable.

One of the perils of working with people is they can type in whatever they want. If someone typed in "one" instead of "1", you'd see an error like this:

```
Traceback (most recent call last):
  File "<pyshell#C>", line 1, in <module>
    int(guess)
ValueError: invalid literal for int() with base 10: 'one'
```

This error is saying you haven't given the `int` function a valid string that is a number it can convert to an integer.

If you compare the logic you want to create in your code with the program so far, you can see that you've checked off a couple of parts (see figure 4.9).

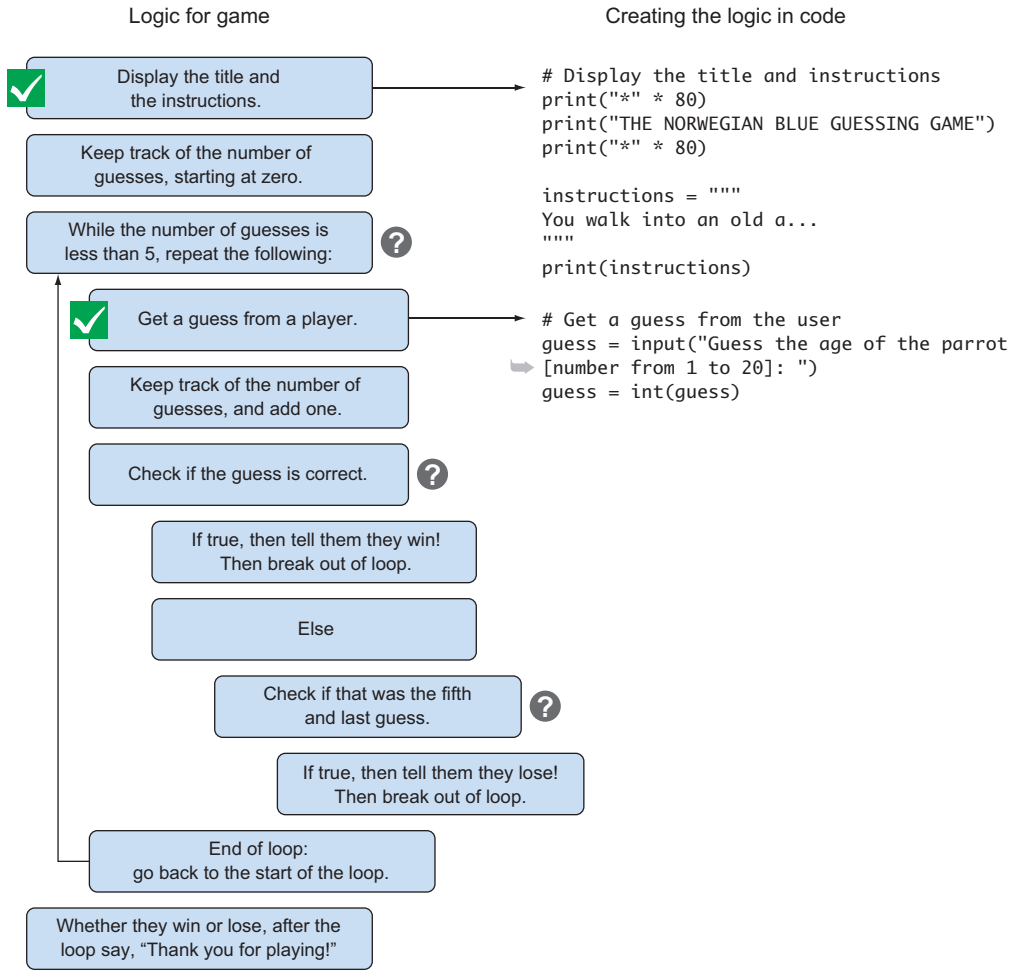


Figure 4.9 On the left is the logic you want to create. On the right is your code. So far, you've welcomed the user and given them the game instructions. You've also added code to collect their guess.

Fabulous! Test the program again to make sure it's working. It'll now ask you to enter a guess. In the next section, you'll see how to test whether the guess is correct.

Using if statements to respond to users in different ways

When you wake up for breakfast, you might walk into the kitchen and look around to see what there is to eat. You use logic to pick your breakfast. If your favorite food is in the kitchen, you'll eat it. For example, if your favorite food is chocolate chip muffins, and there are some in the kitchen, then you'll eat them. If there aren't, you might have a bowl of cereal. In this example, you apply simple logic—you use reasoning to make a decision.

Computer programs use similar logic to interact with users and the world around them. The interactions are based on a set of rules that you (the programmer) write. One of the ways we as programmers can create this logic is with something called the `if` statement.

In your game, you want to test whether the player's guess matches (is equal to) the parrot's age. The logic you want to create in your code is as follows:

- If the player's guess is equal to the parrot's age, congratulate them and give them the Norwegian Blue to take home. End the game.
- Else (if the player's guess isn't equal to the parrot's age) display a mildly insulting message that they're wrong. If it's not their last guess, let them guess again. If it's their last guess, end the game.

Let's use an `if` statement in the program to create the logic you need.

Listing 4.6 Adding logic to the game with an `if` statement

```
# Checking to see if the guess is correct
if guess == parrot_age:
    print("Congratulations! You win!  Enjoy your Norwegian Blue!")
else:
    print("Wrong! You obviously don't know your Norwegian Blues!")
```

Check if guess and parrot_age are equal.

If True, this message displays on the screen.

If False, display a different message.

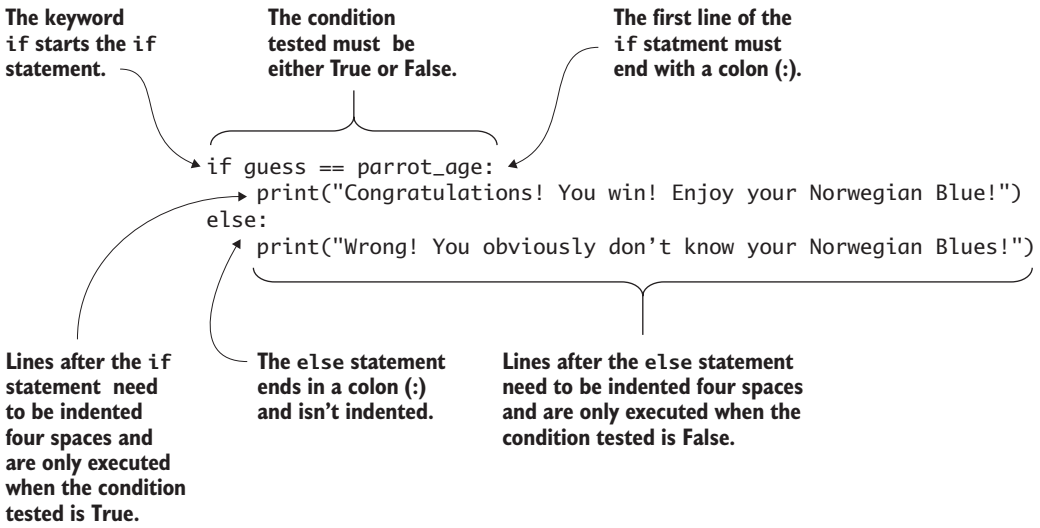


Figure 4.10 The `if` statement can control the flow of your programs. This example shows how an `if` statement can be used to display one message if `guess` is equal to the parrot's age or a different message if they aren't equal.

Let's take a close look at how the `if` statement works and how it gives you a way to create logic in your programs (see figure 4.10).

The keyword `if` is followed by `guess == parrot_age`, and the line ends with a colon (`:`). `guess == parrot_age` is the condition that is being tested. The double equals sign (`==`) is a special operator that checks the equality of `guess` and `parrot_age`.

> TIP Make sure you don't use a single equals sign when testing equality. Single equals signs are used to assign (or store) values into variables.

If they're equals, the `if` condition is evaluated as True, and Python will execute the indented commands after it. In this case, you're printing a message:

```
Congratulations! You win!  Enjoy your Norwegian Blue!
```

If the guess is wrong (`guess == parrot_age` is False), then Python will do the `else` part. The statements to be executed for the `else` part are

indented four spaces. In this case, if the guess is wrong, the program displays this on the screen:

```
Wrong! You obviously don't know your Norwegian Blues!
```

If you examine the code and think back to the logic you want to create, you can see how the `if` statement lets you check whether the guess is correct (see figure 4.11).

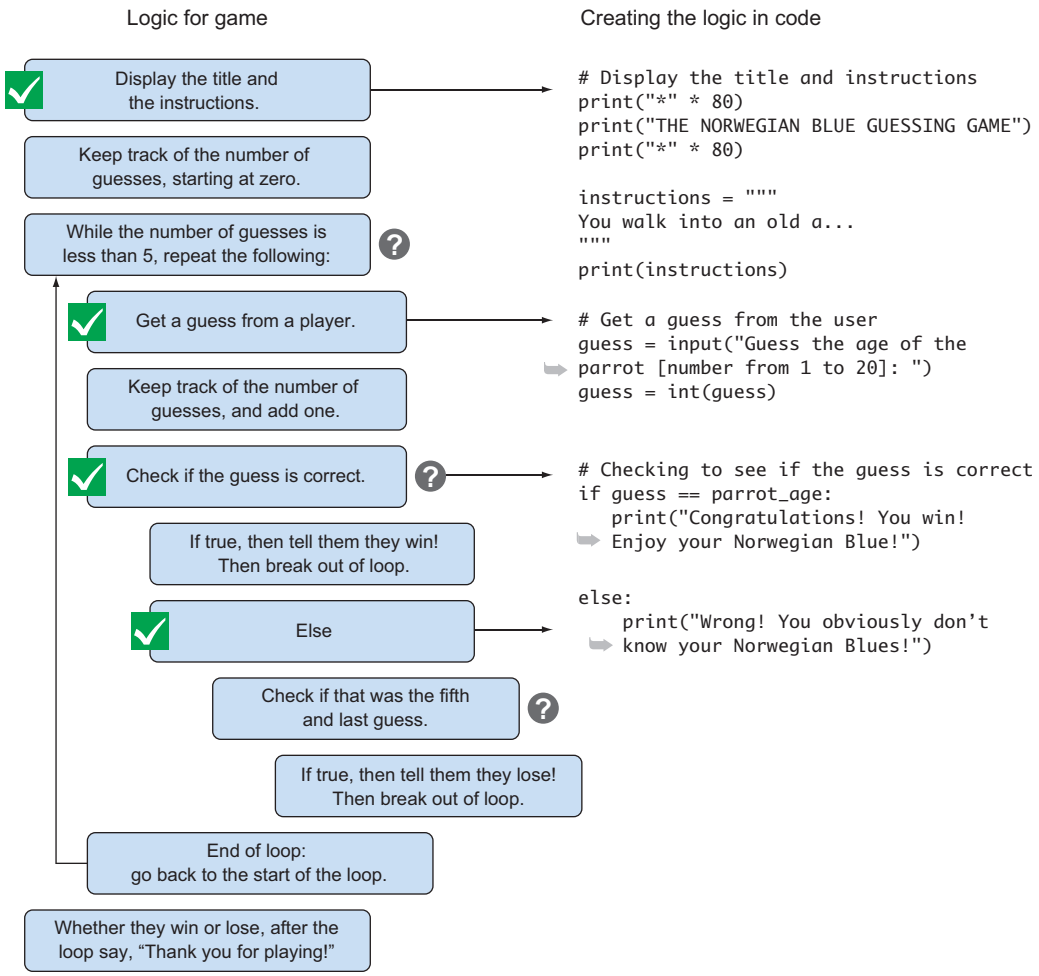


Figure 4.11 The logic you want to create is shown in the code. You use the `if` statement to check whether the player's guess is correct.

You've seen how the `if` statement can make a program make a decision. It's an easy way to control programs by checking whether something is True or False.

There is no "Ummm... maybe"

The `if` statement uses something called *Boolean logic*. In Boolean logic, the answer must always be True or False. There is no "Ummmm... maybe." It's always either True or False.

Boolean logic has its own set of operations for comparisons. These comparisons should be familiar from math class, such as less than (`<`) and greater than (`>`). Here is a table of some of the common comparisons you may need to use with your `if` statements:

Comparison operation	Definition
==	Equal
!=	Not equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

For this game, you're using the equality comparison to check whether two values are equal to each other.

If you need to reverse the logic in a comparison, you can use the `not` operator. The `not` operator changes a True to False or a False to True. If `x` is True, then `not x` is False.

Keep these comparison operators in mind. No matter which one you use, Python analyzes the comparison and returns either a True or False answer.

Practicing if statements

Trying more examples of `if` statements will help you get used to the logic and how to write them. Let's do an example that checks to see

whether a secret password is correct. If it is, the code should grant the person access; otherwise it should deny them access.

Listing 4.7 Using an `if` statement to check a password

```
password = "cheese"
user_password = input("Enter the password: ")
if user_password == password:
    print("Access granted!")
else:
    print("Access denied!")
```

The equal-to comparison checks if the passwords are equal.

The else part executes if the passwords don't match.

Python's `if` statements are a powerful tool for creating programs that respond the way you want them to. You now have the ability to make logic so your programs react and respond based on interacting with a user. This is the first step in adding a bit of artificial intelligence to your programs. Fabulous job!

One of the most common mistakes when working with `if` statements is forgetting to put the colon (`:`) at the end. Figure 4.12 is an example of an `if` statement missing the colon.

Errors are common when writing programs. Try to remember to add a colon at the end of your `if` statements. If Python throws a syntax error box and highlights a space at the end of an `if` statement, you know what you've done.

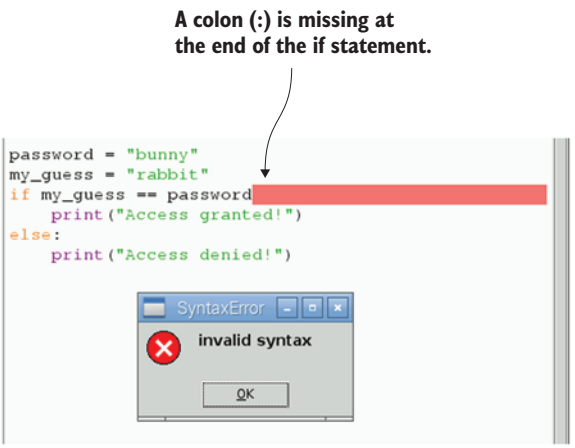


Figure 4.12 A missing colon at the end of an `if` statement will create a syntax error. Python highlights where the error is located. Add a colon at the end of the line to fix the problem.

Using while loops to repeat things

You have input from the user, but you need a way to let the user repeatedly guess the parrot's age. You might get bored repeating something over and over again, but computers will happily repeat something as many times as you want. The repeating parts of programs are called *loops*.

In the case of your guessing game, you're giving the game player five tries to guess the parrot's age. Python has several types of loops: you'll use the `while` loop. A `while` loop repeats over and over until a certain condition or circumstance is no longer true. What it repeats is for you to decide. Each time through the loop, before the program repeats the instructions you gave it, it checks that condition.

Let's look at how you can use a `while` loop with your `if` statement to give the user only five guesses. To help, you'll create a variable named `number_of_guesses` to keep track of the guesses.

Listing 4.8 Using a `while` loop to repeat instructions

```
number_of_guesses = 0
# While loop will repeat until the number_of_guesses is five
while number_of_guesses < 5:
    # Get a guess from the user
    guess = input("Guess the age of the parrot [number from 1 to 20]: ")
    guess = int(guess)

    # Add one to our guess counter
    number_of_guesses = number_of_guesses + 1

    # Checking to see if the guess is correct
    if guess == parrot_age:
        print("Congratulations! You win!  Enjoy your Norwegian Blue!")
        break
    else:
        print("Wrong! You obviously don't know your Norwegian Blues!")

        # Check to see if this is the fifth guess
        # If True, tell them they lost and reveal the parrot's age
        if number_of_guesses == 5:
            print("You lose!")
            print("The Norwegian Blue is " + str(parrot_age))
```

Start the while loop.

Increase the guess counter by one.

Exit the while loop when the answer is correct.

Check if it's the fifth guess and display the end message if it is.

```
    # Stop Indenting (This marks the end of while loop)

print("Thank you for playing!")
```

Notice how you have to rearrange the code in the program a bit. First you start the `while` loop, and then you ask the user to input their guess. Also notice that the code to repeat in the `while` loop is indented (shifted over four spaces). Let's take a closer look at the key elements of the `while` loop (see figure 4.13).

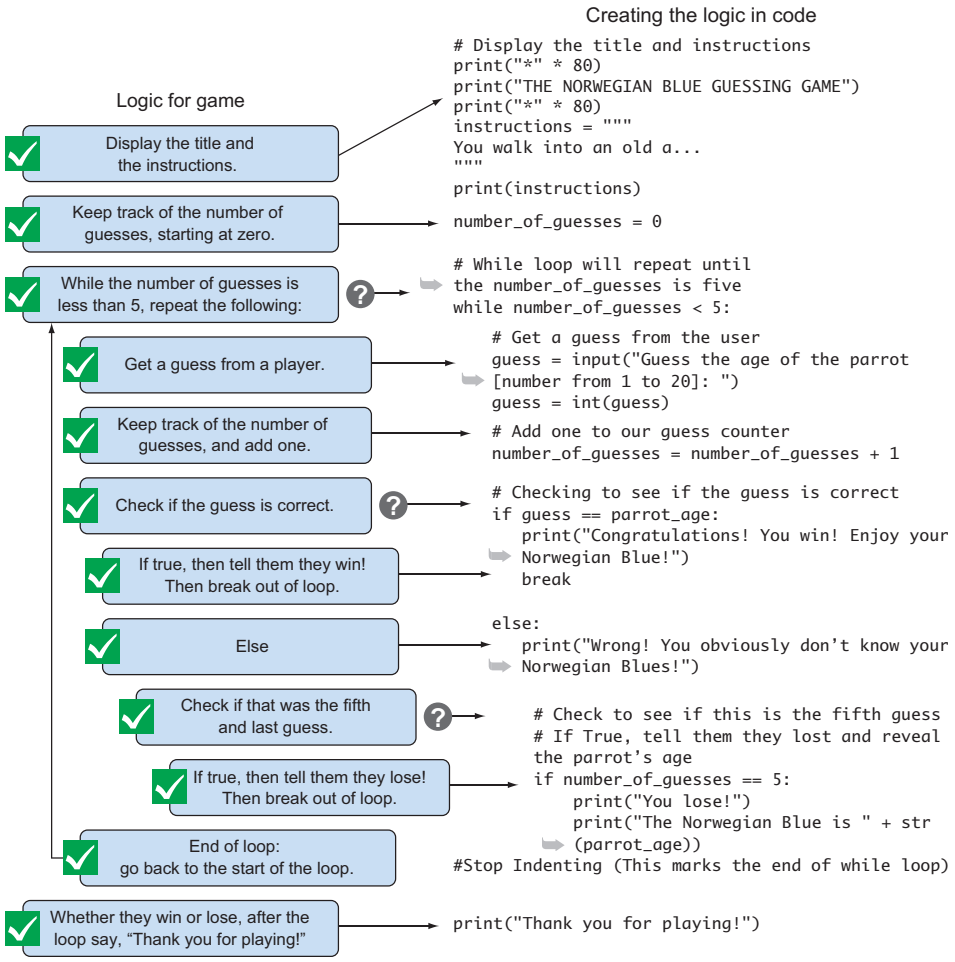


Figure 4.13 Think of the logic you're trying to create, and then translate it into your code. When you need to repeat something, you can use a `while` loop. When you need to check whether something is True or False, you can use an `if` statement.

There is a special thing about a `while` loop: you must indent all the code that you want the loop to repeat (like you did for `if` statements). Each line should be indented four spaces from the left (you measure this from where you type the *w* in `while`). Similarly, you stop indenting code when you want the `while` loop to end. Code that isn't indented is outside the `while` loop and is only run after the `while` loop finishes.

> TIP The IDLE text editor automatically indents the loop text for you. Indentation is used in Python to group code together.

Notice that you create a variable named `number_of_guesses` that helps keep track of how many guesses have been made. It starts with a value of 0; after each guess, the value increases by one. When it reaches a value of 5, if the last guess is incorrect, the game should end. As long as the number of guesses is less than five, the program will check the guess entered by the player to see if it's correct. If a guess is correct, the game should congratulate the player, break out of the loop, and end.

A closer look at while loops

`while` loops run a set of instructions or code repeatedly, but only *while* the condition of the `while` loop is True. This is useful when you want to have something repeat but need a switch that signifies when it should stop. A very common use of `while` loops is in games. A loop makes it so the user can play the game again and again until they say they don't want to play anymore.

The while loop in figure 4.14 counts from 0 to 99. Let's look more closely at its parts.

Like an `if` statement, a `while` loop has an expression that must be either True or False. The example in figure 4.14 uses `count < 100`. The line ends with a colon (`:`), and subsequent lines that belong with the loop should be indented four spaces. In a `while` loop, you can use any other commands you would normally use in Python. To signify the end of the loop, stop indenting statements. Notice that the `print("I finished counting!")` isn't indented, so it's only printed once, after the counting is

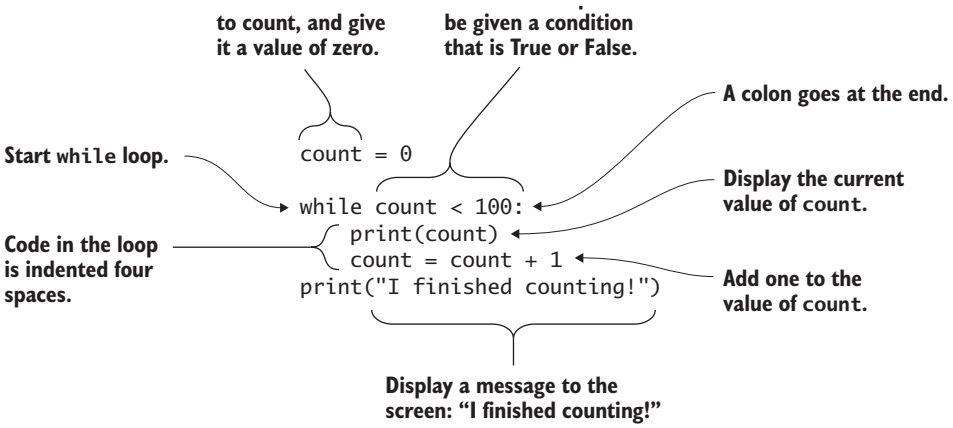


Figure 4.14 You can use a `while` loop to repeatedly perform a task. Code that is part of the loop is indented four spaces. In this case, this `while` loop displays the numbers from 0 to 99; when completed, it prints the message, "I finished counting!" Typically, the condition should be such that code in the loop can make it False and thus end the loop.

complete. Python reads the indentation to know when you want your loop to start and end.

> TIP You can use `if` statements in `while` loops. In your game, you use an `if` statement in a `while` loop. Sounds fancy, but you want to check whether the player's guess is correct, and you need to do this repeatedly to give them their five guesses.

Using loops can save you from writing a lot of code. They let you order a computer to repeat a series of commands many times. The commands only need to be written once in the loop.

Breaking out of a while loop

Sometimes you need to take a break to eat some food or grab a drink. Python has a `break` command that lets you break out of a `while` loop early. In this example, you want your loop to repeat if the player's last guess was incorrect. If the player guesses the parrot's age correctly, then you want to break out of the loop—even though you haven't

reached the fifth guess, you want to stop looping because the player got the answer right.

Let's modify the previous example of counting to 99 so it breaks out of the loop when it reaches the number *77*. You'll use an `if` statement to do this.

Listing 4.9 Breaking out of a loop

```
count = 0
while count < 100:
    print(count)
    if count == 77:
        break
    count = count + 1
print("I finished counting!")
```

Check if the value of count is equal to 77.

If the value is equal to 77, the break command breaks out of the loop.

Practicing while loops

Let's try another example of using a `while` loop to get the hang of how to write them: a `while` loop that asks your favorite color. See if you can figure out what this program does.

Listing 4.10 Favorite colors

```
favorite_color = input("What is your favorite color? ")

while favorite_color != "blue":
    print("Nope, you got it wrong!")
    favorite_color = input("Try again: What is your favorite color? ")

print("Me too! What a coincidence!")
```

This example asks you for your favorite color; if you type in `blue`, it says, "Me too! What a coincidence!" and ends. If you don't input `blue`, the program will keep asking you for your favorite color over and over again (until you say it's blue).

Suppose your loop doesn't produce the output you expect. Maybe the guessing game gives you six guesses instead of five. This is when you try to find the problem and fix it—a process also called *troubleshooting*. Fixing errors in `while` loops can be tricky because there may be

many commands in the loop. The commands execute quickly, so it can be hard to see what is happening. One troubleshooting technique you can use is to add a `print` function in the loop and use it to print out the value of a variable such as the counter each time through the loop.

In this example, you might add this line in your loop:

```
print(number_of_guesses)
```

This prints out the value stored in the `number_of_guesses` variable each time the code goes through the loop. You can see whether the counter is incrementing as you expect and whether it's starting with the right number.

Using Python code libraries to generate random numbers

You program should be working great. The player gets five guesses, and if they guess the age of the parrot correctly, they win! One exciting part about games is their unpredictability—you never know when you might win or lose. Your next task is to have the program pick a random number for the Norwegian Blue's age. This will make it more thrilling because even you won't know the answer!

If you've ever tried to fix a broken bike, toaster, or car, you probably needed some tools. Bare hands are good for many things, but they probably weren't enough for the job. Similarly, in Python, the standard tools (your bare hands) aren't enough. Sometimes you need to get a toolbox and take out a big hammer, soldering iron, or screwdriver.

Python has toolboxes as well. These toolboxes are also called *modules*. Each toolbox (module) contains different sets of tools (methods) that are useful for specific jobs. Here are some examples of common Python modules:

- `datetime` provides useful tools for getting the current time and date and formatting them nicely.
- `random` gives you the ability to create random numbers.
- `math` supports a larger set of mathematical functions.
- `fileinput` supports reading information from files.

Before you can use these toolboxes, you must first carry them into the room, like you might grab a toolbox of bike tools to fix a bike. To bring in a toolbox, you use the `import` command:

```
import random
```

You can add this line anywhere in a program before you need to use it to create a random number. Add it right after the comments at the beginning of your game program. This brings in the toolbox at the beginning of the program and makes it easier for other people who read your code to see what toolboxes (or modules) you're using. What the line is actually doing is loading the toolbox into Python's memory so you can use the tools in your program.

Now that you've added the toolbox, you can use a tool called `randint` to generate a random number between 1 and 20. This code replaces the line `parrot_age = 19`:

```
parrot_age = random.randint(1,20)
```

Notice that you enter the name of the toolbox, put a period or dot (`.`), and then put the name of the tool you want to use. This particular tool, `randint`, needs you to give it two numbers: the lower and upper numbers that the random integer should be between. If you wanted a number between 1 and 100, you'd write

```
parrot_age = random.randint(1,100)
```

With these two lines of code added, the complete code listing should match the code in figure 4.15.

Outstanding! You've made a Norwegian Blue Guessing Game and learned how to create logic in your programs using both `if` statements and `while` loops.

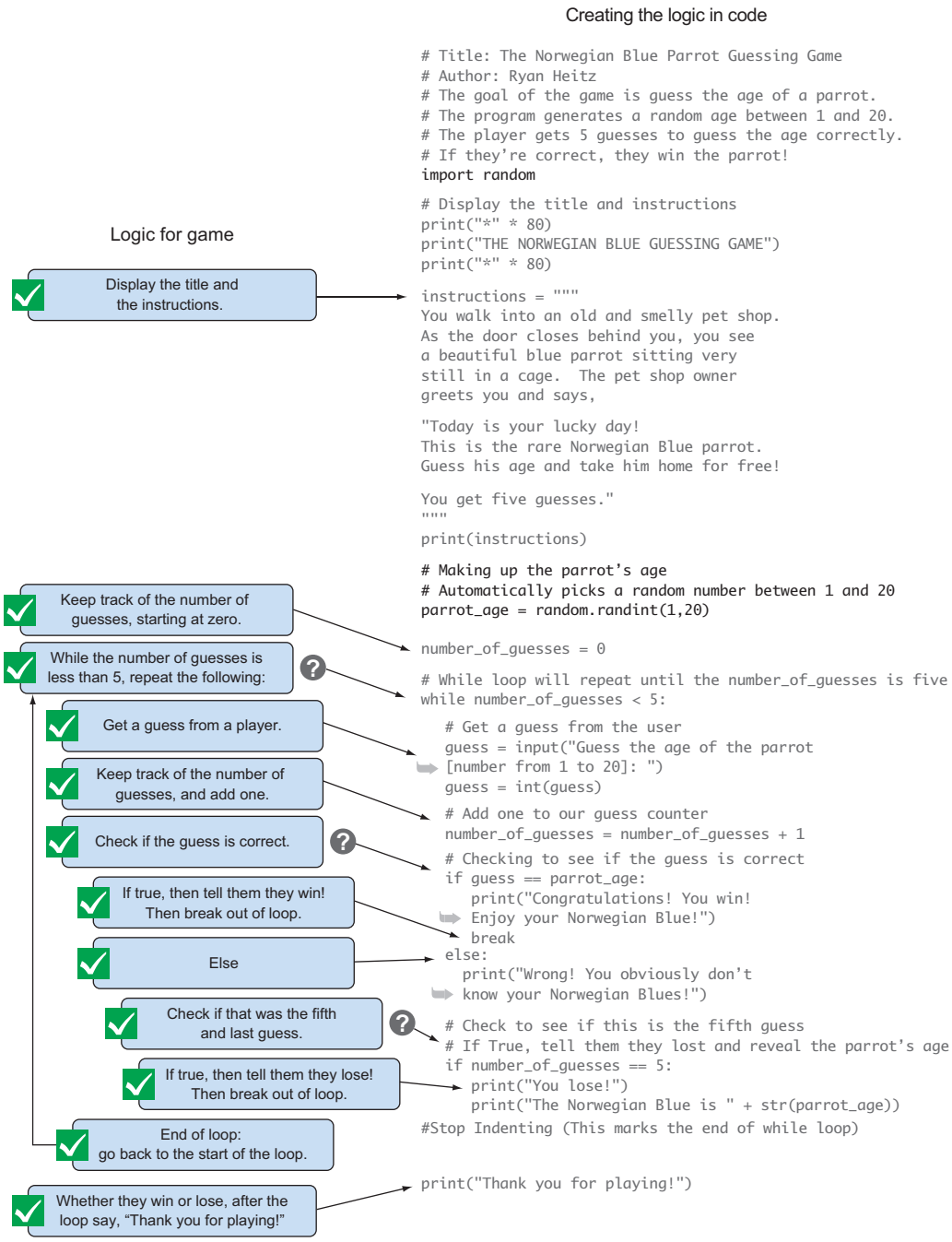


Figure 4.15 To randomly select a number, you need to import the `random` library and use the `randint` function to select a random integer between 1 and 20.

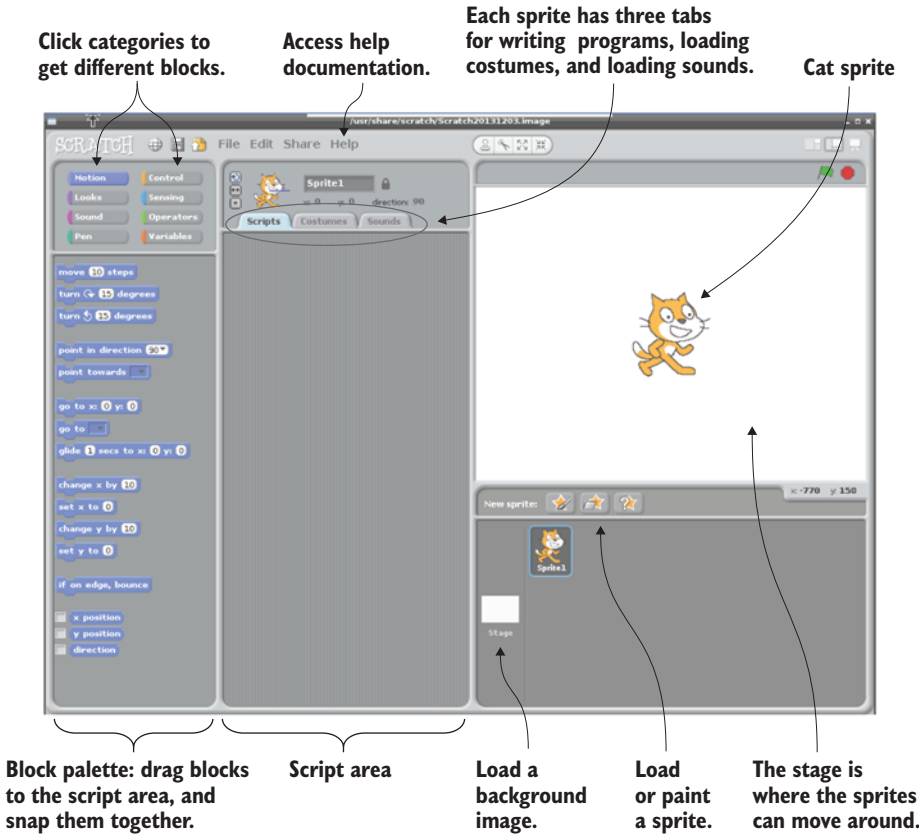


Figure 4.16 The Scratch interface is divided into an area for sprites to move around and a script area. You can create programs for your sprite by dragging blocks and connecting them in the script area.

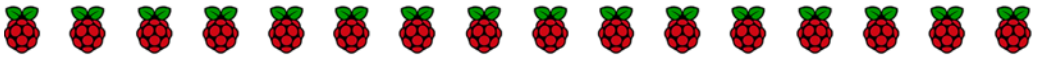
Fruit Picker Extra: Scratch

Have you been wondering why your Pi has an icon that is a picture of a cat head? That is the icon for Scratch. Developed by the Massachusetts Institute of Technology (MIT) to help teach programming, Scratch is a simple program you can use to create animations and games on your Raspberry Pi. Scratch is also its own easy-to-use programming language that is based on dragging and dropping program blocks.

Open Scratch by clicking Menu > Programming > Scratch on your Raspbian desktop. When Scratch opens, you'll see a cat in a white square. Figure 4.16 shows an overview of the Scratch interface.

Scratch can do many things, and we won't explain them all. You can learn more about how to create projects with Scratch by clicking Help > Help Pages. The help tells you how to use each block and provides some tutorials.

Do you have an idea for a project? As in Python, you can make programs that ask for input, display messages, generate random numbers, and use `if` statements and loops. You might add a dog sprite and make it sing like a human when you click it. Or try creating a Scratch version of your favorite classic videogame.



Challenges

Let's play Rock, Paper, Scissors! For this challenge, try to create the classic game.

Rock, Paper, Scissors is played with your hands. Each person simultaneously makes one of three shapes with their hand: the shape of a rock, a piece of paper, or a pair of scissors. If two people make the same shape, it's a tie. The three game shapes interact with each other like this:

- Rock beats scissors.
- Paper beats rock.
- Scissors beats paper.

Let's plan how to attack this challenge. Here are some of the key elements:

- Use a `while` loop to repeatedly ask the player to choose rock, paper, or scissors.
- Create a list of choices:

```
choices = ["Rock","Paper","Scissors"]
```

- Use the `random` library to have the computer randomly choose among the three choices ("Rock", "Paper", and "Scissors").

- Remember, `randint` selects a random integer. You can select and store the random choice in a variable:

```
computer_choice = choices[random.randint(0,2)]
```

- You can select different items in the list by using a number representing where the item is in the list. This number is called a *list index*. In this case, there are three items in the list. The first item has an index of 0, the second item has an index of 1, and the third item has an index of 2. To display the second item in the list, you write `print(choices[1])`; the code displays "Paper" on the screen.
- Use an `if` statement to compare the player's choice to the computer's choice and let the player know who won.
- Ask the player if they want to play again. If so, the loop should repeat; if not, the game should end.

See if you can come up with a program! See appendix C for solutions.

Summary

In this chapter, you've learned some new techniques for working with text in Python and a few foundational elements for creating logic in your programs:

- You can make Python print things just how you want them. String literals allow you to create text that spans multiple lines. Use them to make text appear the same way you typed it in your programs.
- You can write intelligent code that can make decisions. `if` statements add logic to programs by responding only if a certain condition is True. You can combine `if` with `else` statements to make a program do something different if the condition is False.
- You don't have to type things repeatedly—you can make Python repeat them for you. `while` loops can be used to repeat things over and over, as long as a certain condition is True. The `break` command lets you exit a `while` loop if you need to.
- You can use modules (toolboxes) to access more powerful tools to use in your programs. The `random` module has a tool that generates random integers.

5

Raspi's Cave Adventure

In this chapter, you'll create a game to learn new programming techniques:

- *Drawing flow diagrams to map out complex programs*
- *Using Boolean operators to check input from users*
- *Making code for multiple choices using* `if`, `elif`, *and* `else` *statements*
- *Creating and using your own functions to organize code and avoid repeating code*
- *Nesting* `if`/`else` *statements to create games with complex logic*

Like a great book, a game can create an entire imaginary world in your mind. One of the most exciting aspects of games is when you feel like you're inside the game. This doesn't require virtual-reality goggles or high-definition graphics. You can create this immersive feeling even in a completely text-based game by connecting with the player's imagination and creating a world where they can make decisions and determine their own fate. To create games with imaginary worlds, you often have to generate a sense of depth by having the user move from room to room or scene to scene. The game should allow the user to choose their own path

and introduce elements of surprise. Finally, you should also have some great descriptions that make the player feel like they're in the room.

In this chapter, you'll create just such a game, based on exploring an underground cavern. Along the way, the player will have to make choices, and if they make a wrong decision, the game is over. If they make the right decision, they'll find untold treasures of gold, rubies, and diamonds!

Project introduction: Raspi's Cave Adventure

The game is set in medieval days: a time of stone castles, knights with swords, and (some say) mythical beasts that breathe fire. Your main character is a young boy named Raspi.[1] One day Raspi is out gathering firewood and gets lost in the forest. He stumbles upon the entrance to a cave. He peers in the entrance and finds that the cave splits into a left tunnel and a right tunnel. He remembers a folk tale his grandmother used to tell of a mysterious cave in this very forest that holds enormous treasures. It's said the treasure is guarded by a ferocious fire-breathing dragon. Raspi can't resist the temptation to explore the cave; although he knows he should turn back, he walks slowly into the dark cavern. This is the start of your next project: Raspi's Cave Adventure.

The game can have many different outcomes, depending on the path the player chooses for Raspi. A short sample of the program's output is shown in figure 5.1.

Figure 5.1 Raspi's Cave Adventure requires the player to make decisions about which way to go. Based on their choices, the player will meet different fates.

[1] Because this is your game, feel free to make Raspi a girl or a boy.

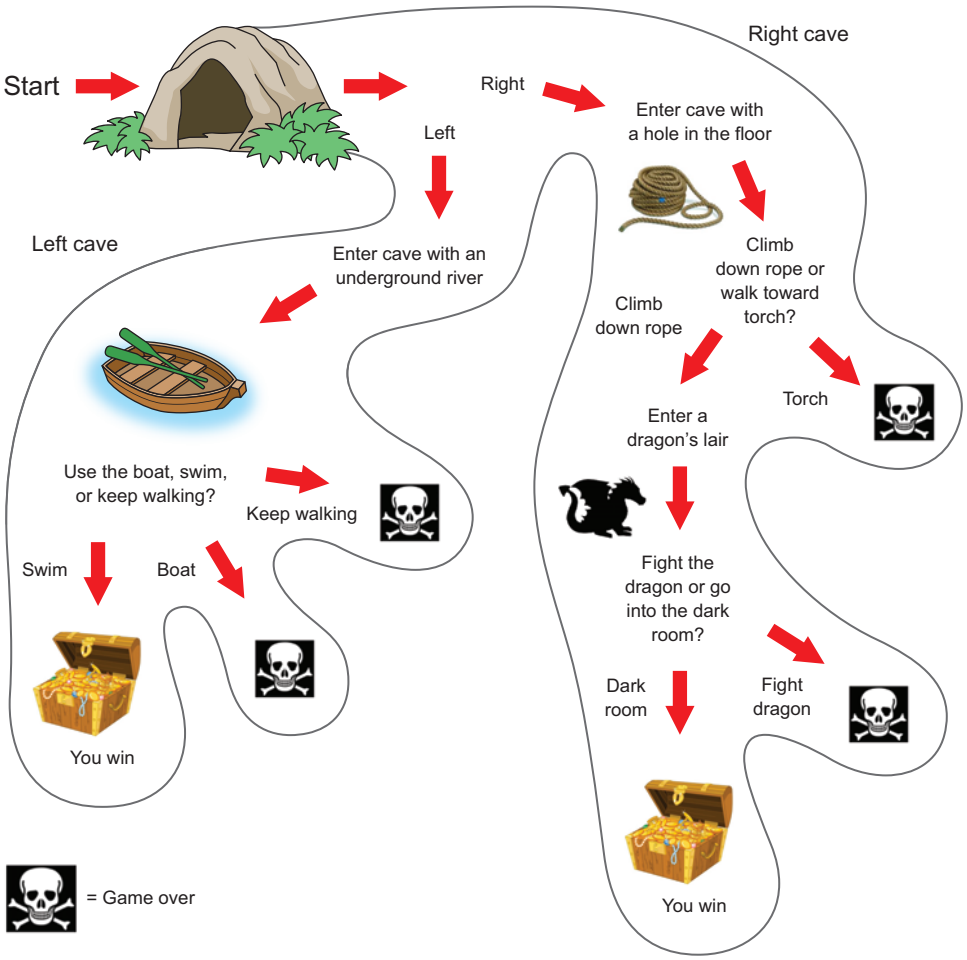


Figure 5.2 This map of the cave system shows that Raspi will need to make many choices. If he makes the wrong ones, it's game over! But if he makes the correct choices, he'll find the legendary treasure!

Let's look at a map of the cave to see where the treasure is and also where the dragon lives! Because you're the game designer and developer, you'll use this as a guide to write the code creating the game logic (see figure 5.2).

Let's examine the different paths and choices Raspi has in the cave and his possible fates. After Raspi enters the entrance to the cave, he can choose to go left or right.

Left cave

If Raspi goes into the left cave, he'll find himself near an underground river. He'll need to decide whether to take a boat down the river, swim down the river, or walk along the side of the river. If Raspi decides to take the boat, he'll soon learn that it has a hole in it, and he'll sink (game over). Should Raspi choose to avoid the river and walk along its edge, he'll quickly become distracted by his thoughts, trip on a rock, and hit his head (game over). If Raspi is adventurous and decides to swim in the river, he'll make it to the other side and find a hidden treasure room filled with riches!

Right cave

If Raspi decides to go into the right cave, he'll need to decide whether to climb down into a hole using a rope or walk toward what appears to be a torch. After walking toward the torch, Raspi will enter a cave full of crystals. The crystal cave sounds promising, but unfortunately a crystal will fall from the ceiling, ending Raspi's life (game over). Alternatively, if Raspi uses the rope and goes down the hole, he'll find himself in the dragon's lair with a final choice: whether to fight the dragon or go into a dark room. If Raspi fights the dragon, the dragon will eat him; but if Raspi heads toward the dark room, he'll discover that it's filled with thousands of gold coins, rubies, and diamonds. Raspi is rich and very much alive!

Hey wait, you need a plan (flow diagrams)

Your goal is to create a program that allows the player to make multiple decisions. You have a map of the cave; now you need to make that map into a diagram that can guide you as you write the code for the game. Much as you did in chapter 2, you'll lay out the logic of the game and then write the code to create that logic.

You can make a map that also functions as a flow diagram. You can visualize the set of decisions and the outcome of each decision. Figure 5.3 shows the map of the cave as a flow diagram.

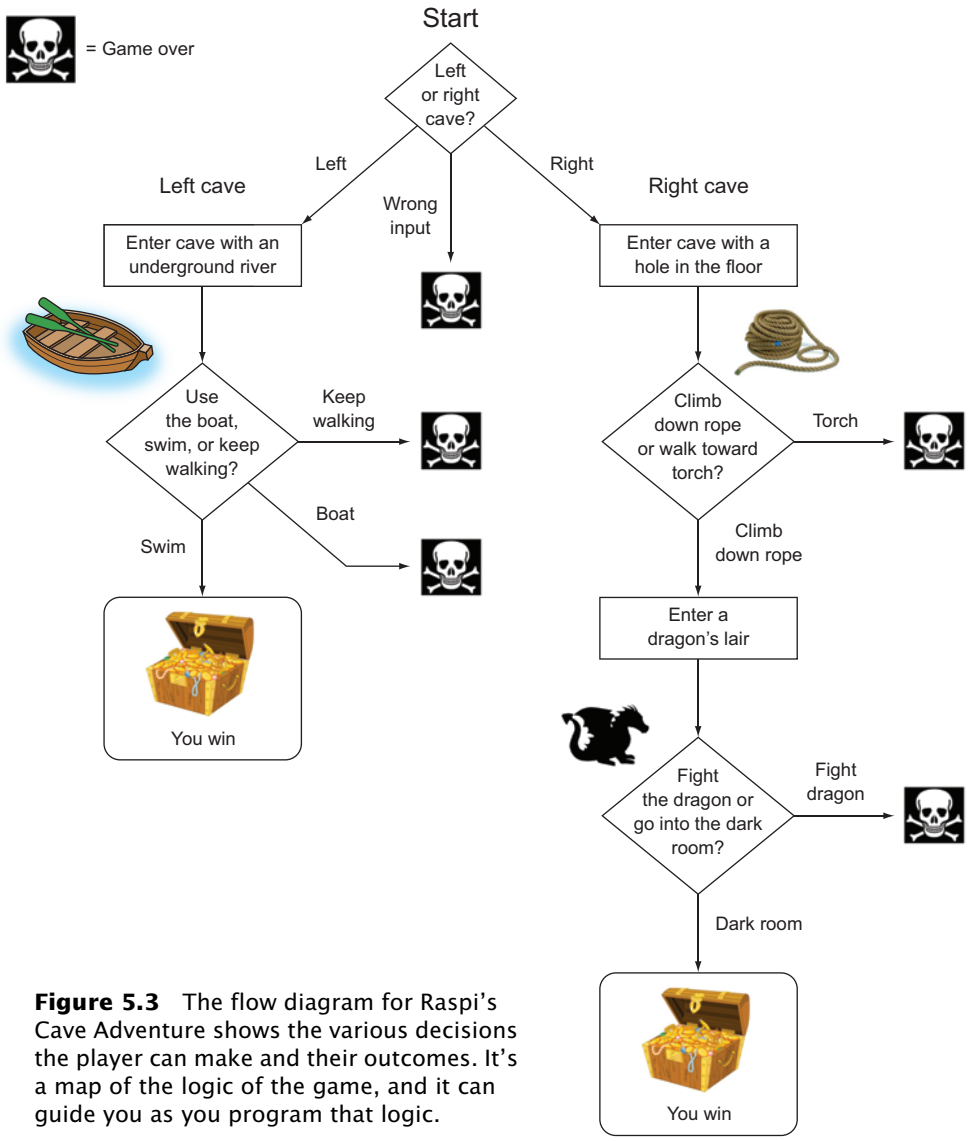


Figure 5.3 The flow diagram for Raspi's Cave Adventure shows the various decisions the player can make and their outcomes. It's a map of the logic of the game, and it can guide you as you program that logic.

Each decision in the diagram is represented by a diamond shape. Inside the diamond is the question at hand. Outside the diamond are arrows representing the possible choices available and the result of each choice. Sometimes choices lead to other choices (other diamonds). Other times, a choice leads to winning the game or game over!

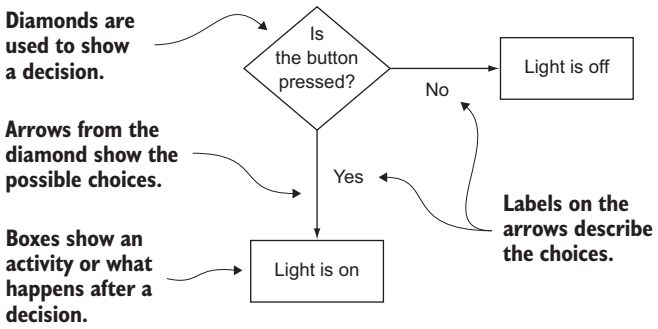


Figure 5.4 Flow diagrams are ways to visually show the logic of a program. They represent decisions, choices, and activities using diamonds, arrows, and boxes. This example shows a flow diagram for a program that turns on a light if a button is pressed.

Flow diagrams follow a few simple rules (see figure 5.4). You can construct one for any set of decisions, including those used by games, robots, and apps.

A flow diagram is a great way to organize your thoughts and break down complex problems into a series of simple steps. Remember the Python way: simple is better than complex.

Which way should Raspi go? (checking input)

With your diagram in hand, the first bit of logic is the user choosing whether to go left or right. Let's display text to tell the player what they see in the cave, and then prompt them to enter a choice. You prompt the user and collect information with the `input` function.

Listing 5.1 Choosing the left or right cave

```
# 1st Choice: Left or Right Cave?
print("You see the cave split into a left and right tunnel")
print("Do you choose to go left or right?")
cave_choice = input("Enter L for left or R for right: ")
if cave_choice == "L":
    # Left cave
    print("You walk into the left cave.")
else:
    # Right cave
    print("You walk into the right cave. The cave starts sloping downward.")
```

The input function asks the user to enter a choice and stores it in a variable cave_choice.

Display descriptive text to the screen.

Check if cave_choice is equal to L.

If True, display text about walking into the left cave.

If False, display text about walking into the right cave.

This example uses the `input` function and then an `if`/`else` statement to create the logic you want. The code asks the user to make a choice by

typing `L` or `R`. The `if` statement checks whether the user's choice equals "L". If True, then the code displays a message that the player entered the left cave. If their choice isn't equal to L (if that condition is False), then the program moves to the `else` statement and displays a message that the player entered the right cave.

Handling unexpected input

Users often do unexpected things. As a programmer, one thing you have to be thinking about is what happens if the user does something you don't expect. The person playing your game can type in whatever they want. Let's examine some different possibilities and see what would happen:

- What if the user types in `l` (lowercase `L`)?

  If the user types in `l`, the program checks (evaluates) whether "l" is equal to "L". Because these two strings are different, this condition is False. The program will execute the `else` statement and display a message that the user entered the right cave.

- What if the user types in `left`?

  If the user types in `left`, the program evaluates whether "left" is equal to "L". Because these two strings are different, this condition is False. The program will execute the `else` statement and display a message that the user entered the right cave.

- What if the user types in something like `44992` or `banana` just to be silly?

  The program checks whether "44992" or "banana" is equal to "L". Because neither of these equals "L", this condition is False. The program will execute the `else` statement and display a message that the user entered the right cave.

- What if the user enters anything except `L`?

  You guessed it; they will see a message that they entered the right cave.

This isn't ideal. Let's improve the code as follows:

1. Permit the user to enter `L` or `l` as well as `Left` or `left` to enter the left cave.
2. Permit the user to enter `R` or `r` as well as `Right` or `right` to enter the right cave.

3 Take care of anything else by having the game scold the user for entering the wrong thing and end the game in a humorous way. Maybe a stalactite could fall from the ceiling or a cave spider could bite them!

To create this behavior, you need to introduce the Boolean `or` operator. You also need to convert the input information to all uppercase letters using Python's `upper()` method. Finally, to handle all three possible outcomes, you'll use a new `if`/`elif`/`else` statement (see listing 5.2).

Methods

Methods are functions that only work on specific types of Python things, which programmers call *objects*. In this example, `.upper()` is only able to work on strings, so it's called a *string method*. Methods are called differently than other functions. Methods use *dot notation*, which means you type the name of the thing (object) and then put a dot (`.`) and the method.

Here are some examples:

- `"Left".upper()` produces "LEFT".
- `"riGHt".lower()` makes "right".

Here's the updated code to apply these new ways to avoid errors in user input.

Listing 5.2 Improving the code for the player's choice

```
# 1st Choice: Left or Right Cave?
print("You see the cave split into a left and right tunnel")
print("Do you choose to go left or right?")
cave_choice = input("Enter L for left or R for right: ").upper()
if cave_choice == "L" or cave_choice == "LEFT":
    # Left cave
    print("You walk into the left cave.")
elif cave_choice ==  "R" or cave_choice == "RIGHT":
    # Right cave
    print("You walk into the right cave. The cave starts sloping
    ➥ downward.")
else:
    # Wrong answer
    print("You seem to have trouble making good decisions!")
```

Gather input from the user. The upper() method converts the user's input to all uppercase.

The or operator checks if either condition is True.

elif checks if another condition is True.

else handles the case where all if or elif statements are False.

```
print("Suddenly a stalactite falls from the ceiling and bonks you
➥ on the head.")
print("Game Over!!!")
```

The `upper()` method converts the input text to all uppercase. If the user enters `LEFT`, `LeFt`, `left`, or `Left`, the string is converted to "LEFT".

THE BOOLEAN OR OPERATOR: CHECKING WHETHER EITHER ONE IS TRUE

The `or` operator checks whether one condition or another condition is True. This gives your code more flexibility—it's able to accept more than one input and still proceed. If either one is True, the `if` statement is True, and Python does whatever is indented under the `if` statement.

ELIF IS SHORT FOR ELSE IF

The `elif` statement is short for `else if`. It checks whether another condition is True. Think of it like a multiple-choice question. If the user doesn't enter `L` or `Left`, the program moves on to the next option. If the user doesn't enter `R` or `r`, the program moves to the `else` statement and drops a stalactite on their head. Game over! Take a closer look at the `if`/`elif`/`else` statement in figure 5.5 to see how to make one.

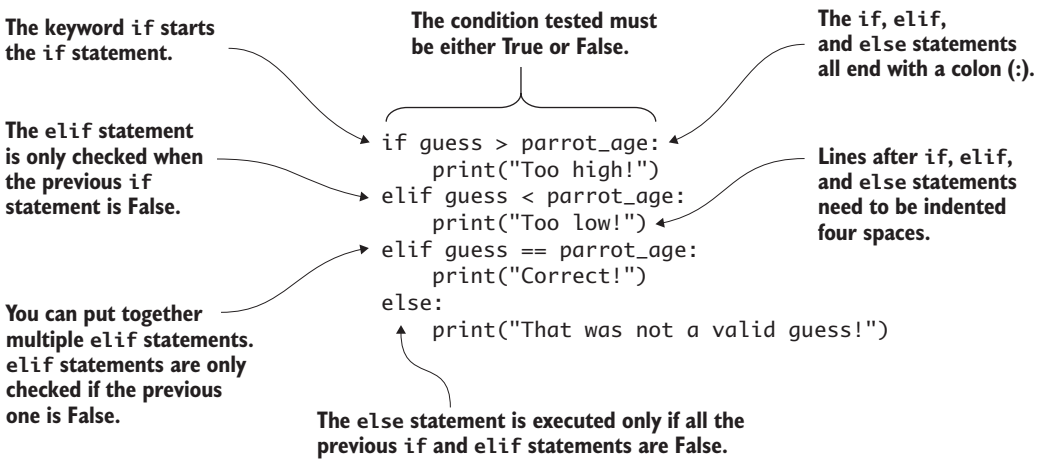


Figure 5.5 The `if` statement can come in many flavors. This is an `if` statement with two `elif`s and an `else`. It creates logic in the code that can do many different things depending on the user's input. In this case, you're having a player guess the age of a parrot. The program will tell them if their guess is too low, too high, correct, or invalid.

Notice that you can have more than one `elif` statement. In fact, you can have as many as you want. With the `if`/`elif` statement, you can create the logic needed for your cave.

Boolean logic operators: `and`, `or`, and `not`

Python has a complete set of Boolean operators that you can use to make expressions:

- `or` is used when you want the expression to be True if either of the operands is True.
- `and` is used when you want the expression to be True only if both operands are True.
- `not` is used to change an operand from True to False or False to True.

Let's look at a few examples using these operators.

`and` OPERATOR

Pretend you want to create a program giving you access to the system only if your name *and* password are *both* correct. You could write this using the `and` operator:

```
if name == "Ryan" and password == "PiTaster":
    print("The name and password are correct!")
    print("Access granted! Welcome!")
else:
    print("Access denied!")
```

Only if both `name` and `password` are correct will the program grant you access. Try creating one yourself!

`or` OPERATOR

Next let's imagine you want to create a program giving someone a free pizza if their age is under 20 *or* they have a coupon. Let's assume you have a variable `age` that is the age of the person and another variable `coupon` that already holds a value of True or False. Using the `or` operator, you create this logic like so:

```
if age < 20 or coupon == True:
    print("You get 1 FREE PIZZA")
else:
    print("No free pizza for you!")
```

If *either* is True, the user gets a pizza. If *both* are True, they get a pizza. If *neither* is True, then no free pizza!

not OPERATOR

Finally, let's say you have a variable `is_absent` that is equal to True or False. `is_absent` tells you whether a student is present or absent. To print a "Welcome to school!" message if a student is *not* absent, you can use the `not` operator:

```
if not is_absent:
    print("Welcome to school!")
else:
    print("Please return to school as soon as possible. School misses
          you!")
```

The `not` operator changes a variable or statement that is True to False and a False one to True. It helps you create conditional statements (`if` statements) that make more sense when you read the code. As you can see, the Boolean operators give you many different options for creating logical expressions.

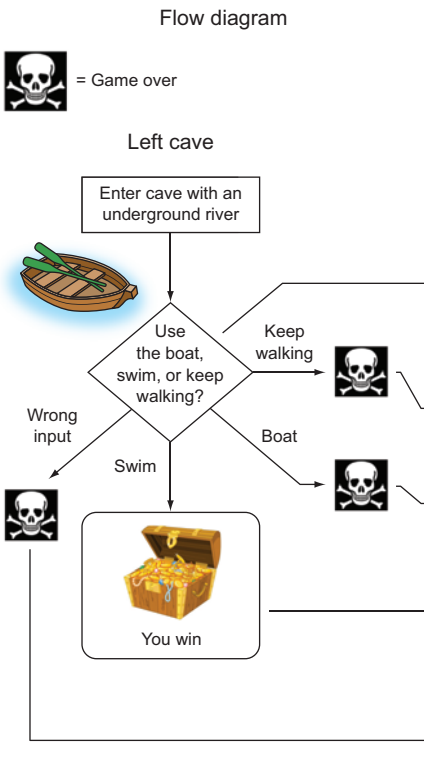
Time to go spelunking (a fancy word for exploring caves) with your new knowledge of `if`/`elif`/`else` and Boolean operators!

Turning flow diagrams into code

For now, let's concentrate on building a program for the left cave. The player has entered the left cave and needs to make their next choice. Looking at the map and the flow diagram, the next thing your player encounters is an underground stream. The player sees a boat and must choose among three options:

- Keep walking along the side of the river.
- Climb into the boat.
- Swim in the river.

Each of these will be an `if` or `elif` statement in your code. But wait! There's a fourth possible outcome—that they don't enter one of the




```
Code
# 1st Choice: Left or Right Cave?
print("You see the cave split into a left and right tunnel")
cave_choice = input("Enter L for left or R for right: ").upper()

if cave_choice == "L" or cave_choice == "LEFT":
    # Left cave
    print("You walk into the left cave.  It is cold and dark.")
    print("The cave opens up to a large room with an
➥ underground river.")
    print("You notice a small boat on the edge of the river.")
    print("Do you use the boat, swim, or walk along the side
➥ of the river?
    river_choice =  input("Enter B for boat, S for swim, or W
➥ for walk: ").upper()

    if river_choice == "W" or river_choice == "WALK":
        # You walk along the edge of the river
        print("You walk along the narrow edge of the river.")

    elif river_choice == "B" or river_choice =="BOAT":
        # You hop in the boat
        print("You step in the boat and start drifting
➥ down the river.")

    elif river_choice == "S" or river_choice == "SWIM":
        # You jump in the water and start swimming
        print("You dive into the water and start swimming
➥ down the river.")

    else:
        # Wrong input
        print("You seem to have trouble making good decisions!")
        print("Suddenly a stalactite falls from the ceiling and
➥ bonks you on the head.")
        print("Game Over!!!")

elif cave_choice ==  "R" or cave_choice == "RIGHT":
    # Right cave
    print("You walk into the right cave.  The cave starts
➥ sloping downward.")
else:
    # Wrong answer
    print("You seem to have trouble making good decisions!")
    print("Suddenly a stalactite falls from the ceiling and
➥ bonks you on the head.")
    print("Game Over!!!")
```

Figure 5.6 The left cave has a stream inside it, and the user has three choices of what to do next. In the code, you create an `if` statement followed by two `elif` statements to cover each of the options. The `else` statement is used to control what happens if the user inputs something other than one of the three choices.

three choices. You'll make this the `else` statement. Figure 5.6 shows the left cave flow diagram and the code that creates the logic you need.

You display a few words about what Raspi sees inside the left cave. You ask the user to choose what to do next. Then, once you've

gathered this input, you evaluate that information and respond accordingly. Notice that each of the possible choices appears in an `if` or `elif` statement and is indented under the left cave `if` statement. The user has to choose whether to keep walking (`W`), use the boat (`B`), or swim (`S`). For each case, the program should display information as you designed it in your flow diagram.

This isn't only for caves

Boolean operators and `if`/`elif`/`else` statements are great for when your program needs multiple options or choices. Let's see if you can create a program that has four possible options: A, B, C, and none of the above. The following snippet shows an example of using `elif` statements to create these four possible outcomes. In this example, you're pretending that a person is on a game show and picking a door with a prize behind it:

```
print("Welcome to the Pi Game Show!")
print("There are three doors with prizes behind them: A, B, and C.")
door = input("Select a door by typing A, B, or C").upper()

#Logic for door selection
if door == "A":
    print("You've won a new car!")
elif door == "B":
    print("You've won a new boat!")
elif door == "C":
    print("You've won a trip around the world!")
else:
    print("Uh oh! You didn't follow directions!")
    print("Game Over!!!")

print("Thank you for playing.")
```

Creating programs with choices based on logic is a powerful programming skill. By combining simple choices, you can create complex programs.

Excellent work! You've created the left cave logic for Raspi's Cave Adventure. Let's add more decisions.

Simplify! Making your own functions

Yikes! The code for the left cave is starting to look long (and kind of ugly and hard to read), and you still have the right cave to go. How can you simplify your program?

The answer is *functions*. This time you aren't going to call a built-in Python function—you'll make your own!

Functions are like mini programs that you can create to organize or simplify your code. When you have long programs, you can take logical chunks of code (code that all goes together) and put them in a function. Once you've created (or defined) a function, you can call (or use) the function in your code.

> NOTE Functions should always be defined at the top of a program. The definition of a function must come before it's called (or used).

Let's see how this works by making (or defining) two functions for the left cave.

Listing 5.3 Creating functions for the left cave

```
# Displays a description of the left cave and their choices
def left_cave():
    print("You walk into the left cave. It is cold and dark.")
    print("The cave opens up to a large room with an underground
    ➥ river.")
    print("You notice a small boat on the edge of the river.")
    print("Do you use the boat, swim, or walk along the side of the
    ➥ river?")
    river_choice =  input("Enter B for boat, S for swim, or W for walk:
    ➥ ").upper()
    return river_choice

# Displays text describing the player's demise and a game over message
def wrong_answer():
    print("You seem to have trouble making good decisions!")
    print("Suddenly a stalactite falls from the ceiling and bonks you
    ➥ on the head.")
    print("Game Over!!!")
```

def defines a function. After def is the name of the function and a colon.

The function's instructions (what it does) must be indented four spaces under the def statement.

Gather input from the user and store it in a variable river_choice.

Send information to the program when the function is called.

Display text for ending the game if the wrong input is given.

Before moving on, let's look more closely at how you can make your own functions (see figure 5.7). You've created two functions: `left_cave` and `wrong_answer`. Let's rewrite the cave program to use (or call) those

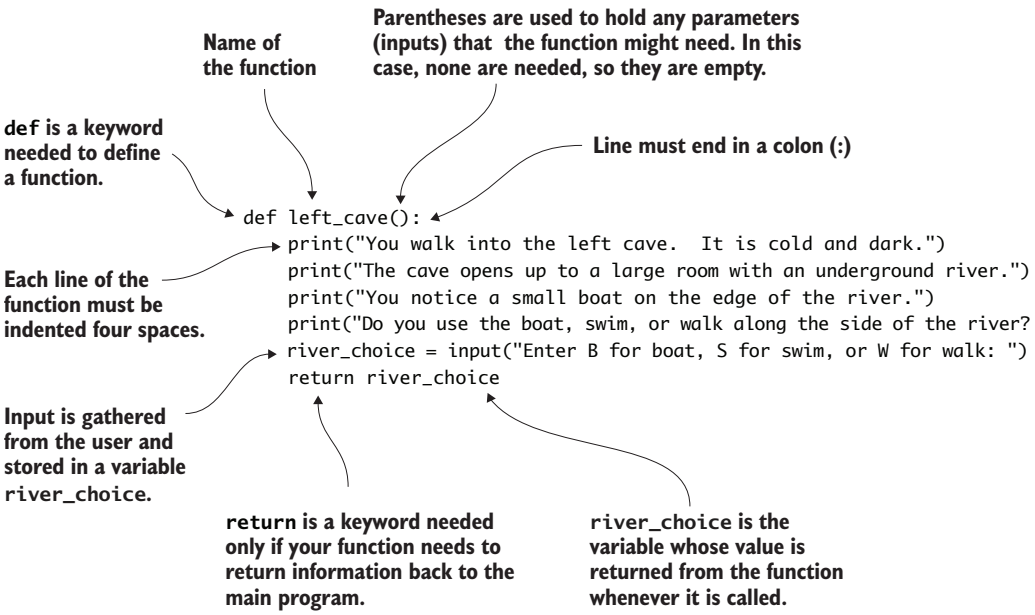


Figure 5.7 Functions simplify your code and can reduce repetition. Use the `def` keyword to create a new function, and indent the function code under it. If you need a function to return a value, include a `return` statement in the function.

functions. Whenever you call a function, it's as if the code is all in that spot, but you've hidden it.

Some functions need to return something; others don't. You might have a function that prints something to the screen or plays a sound; those types of functions don't need to return anything. In the example code, the `wrong_answer()` function is a good example. You call the function like this:

```
wrong_answer()
```

Alternatively, when a function returns something and you want to store that information, you write it like this:

```
choice = left_cave()
```

This takes whatever information is returned by calling the `left_cave()` function and stores it in a variable named `choice`. Listing 5.4 shows how you can simplify the program by calling the `left_cave()` and `wrong_answer()` functions.

Listing 5.4 Using the new functions to simplify your code

```
# 1st Choice: Left or Right Cave?
print("You see the cave split into a left and right tunnel")
cave_choice = input("Enter L for left or R for right: ").upper()
if cave_choice == "L" or cave_choice == "LEFT":
    # Left cave
    choice = left_cave()

    if choice == "W" or choice == "WALK":
        # You walk along the edge of the river
        print("You walk along the narrow edge of the river.")
    elif choice == "B" or choice == "BOAT":
        # You hop in the boat
        print("You step in the boat and start drifting down the
        ➥ river.")
   elif river_choice == "S" or river_choice == "SWIM":
        # You jump in the water and start swimming
        print("You dive into the water and start swimming down the
        ➥ river.")
    else:
        # Wrong answer
        wrong_answer()
    elif cave_choice == "R" or cave_choice == "RIGHT":
    # Right cave
    print("You walk into the right cave. The cave starts sloping
    ➥ downward.")
    print("You come to a room with a large hole in the floor.")
else:
    # Wrong answer
    wrong_answer()
```

left_cave() calls your function. The information returned by the function is stored in the variable choice.

The statements that displayed game-over information are replaced by calling the wrong_answer() function.

wrong_answer() can be called as many times as needed.

Amazing! The resulting code is easier to read, and you avoid repeating code. Notice that you call the `wrong_answer()` function twice. This saves you from having to write those lines of code twice. Also, if you ever

want to change the ending for a wrong answer, you only have to change it in one place (in the function). In addition to helping you organize your code, the ability to reuse functions is one of their key features. You haven't changed the functionality of your program, but by using functions, you've made it easier to read and simplified it.

DEFINITION *Refactoring* is a programming technique that focuses on reorganizing and simplifying code in a program. Refactoring makes the code easier to read and less complex.

Passing parameters: functions with inputs

You've looked at two different functions so far: one that doesn't return anything and one that does. Functions have another feature in addition to their ability to return something—they can also receive information. Think of it as input to a function. In programming speak, you say that the function has a *parameter* or *parameters*. Let's see how this works with an example. Suppose you have a guessing game, and you want to create a function that prints a message to the screen telling the player if their guess is too high, too low, or spot on:

```
def check_guess(guess, answer):
    # Compare the guess to the answer
    if guess == answer:
        print("You're correct!")
        is_correct = True
    elif guess < answer:
        print("Too low!")
        is_correct = False
    elif guess > answer:
        print("Too high!")
        is_correct = False
    else:
        print("Invalid guess")
        is_correct = False
    # Return True or False depending upon if the guess is correct
    return is_correct
```

In this case, the `def` statement has the name of your function (`check_guess`). Inside the parentheses are two parameters separated by a comma: these are inputs to the function. The first input or parameter is `guess`. This is a guess the user has made. The second is `answer`, which is the number the user is trying to guess. The function then compares `guess` and `answer` and tells the user whether they were right or guessed too low or too high. The great thing about this function is that it can work with any numeric guess and answer (1 to 10, 1 to 1,000,000). By using parameters, you make the code more flexible.

The best way to learn about functions is by doing. Here are some functions dos:

- Use a simple name that describes the function.
- Put comments about your function inside the function.
- Return values when you want to use them in a program.

And here are some functions don'ts:

- Use complex names.
- Create functions with only one line of code.
- Forget to put a colon at the end of the `def` statement.
- Forget to call the function in your main program.

Fantastic programming! You're achieving the Zen of Python by simplifying your code with functions.

Finishing the left cave

To complete the left cave, you need to add code for Raspi's choices: walking along the river's edge, taking the boat, or swimming (the winning ending). You'll make each of these choices its own function to help organize your code and keep it uncluttered. You can call the functions in the main program, shown in the next listing.

Listing 5.5 Calling functions for each of the left cave choices

```
# Main Program
# 1st Choice: Left or Right Cave?
    choice = left_or_right()
    if choice == "L" or choice == "LEFT":
        # You walk into the Left cave
        choice = left_cave()
        if choice == "W" or choice == "WALK":
            # You walk along the edge of the river... game over
            walk()
        elif choice == "B" or choice == "BOAT":
            # You get in the boat... game over
            boat()
```

Call a function called walk() that displays messages about Raspi's fate. See the source code for examples of the functions.

Call a function named boat() that tells you what happens if Raspi gets in the boat.

```
        elif choice == "S" or choice == "SWIM":
            # You jump in the water and start swimming... Raspi wins
            swim()
        else:
            # Wrong answer
            wrong_answer()
    elif choice == "R" or choice == "RIGHT":
        # You walk in the right cave
    else:
        # Wrong answer
        wrong_answer()
```

You guessed it: call the swim() function that contains the code for Raspi swimming.

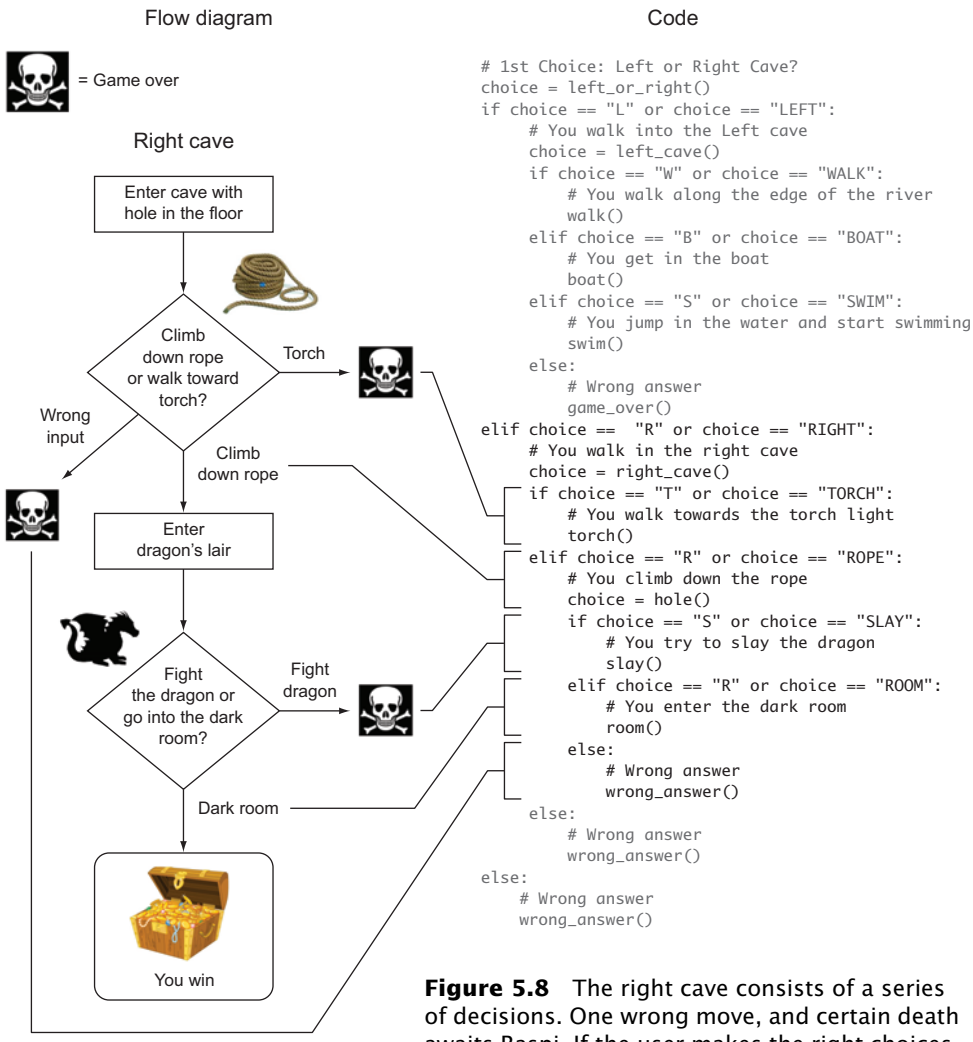
See the source code for chapter 5 for examples of each of these functions (`walk()`, `boat()`, and `swim()`). They follow a structure similar to the `left_cave()` and `wrong_answer()` functions. Feel free to make up your own descriptions of what happens to Raspi or change the outcomes to how you would like them.

Exploring the right cave

In this game, Raspi has two initial cave choices: left or right. Programming the right cave is similar to the left cave. Once again, you'll use the map and flow diagram as your guides. Let's add the logic for the right cave, which starts with the user finding a hole in the ground (see figure 5.8).

The right cave uses logic similar to that of the left cave. You'll use `if`, `elif`, and `else` statements to handle all the possible choices. As with the left cave, notice that you indent the `if`/`elif`/`else` statements under the other `if` statements to create the logic you desire. *Nesting* is the name given to indenting one set of `if` statements inside another. The technique of nesting `if` statements is useful when you have logic that you want executed only if a prior condition is True. In this case, you only want to give the user the choice of fighting the dragon if they have already decided to climb down into the hole using the rope. The logic now matches the flow diagram for the game.

Let's take another look at nesting using a different example. Imagine that you want to write a program that displays a secret message after you



Code

```
# 1st Choice: Left or Right Cave?
choice = left_or_right()
if choice == "L" or choice == "LEFT":
     # You walk into the Left cave
     choice = left_cave()
     if choice == "W" or choice == "WALK":
          # You walk along the edge of the river
          walk()
     elif choice == "B" or choice == "BOAT":
          # You get in the boat
          boat()
     elif choice == "S" or choice == "SWIM":
          # You jump in the water and start swimming
          swim()
     else:
          # Wrong answer
          game_over()
elif choice ==  "R" or choice == "RIGHT":
     # You walk in the right cave
     choice = right_cave()
     if choice == "T" or choice == "TORCH":
          # You walk towards the torch light
          torch()
     elif choice == "R" or choice == "ROPE":
          # You climb down the rope
          choice = hole()
          if choice == "S" or choice == "SLAY":
               # You try to slay the dragon
               slay()
          elif choice == "R" or choice == "ROOM":
               # You enter the dark room
               room()
          else:
               # Wrong answer
               wrong_answer()
     else:
          # Wrong answer
          wrong_answer()
else:
    # Wrong answer
    wrong_answer()
```

Figure 5.8 The right cave consists of a series of decisions. One wrong move, and certain death awaits Raspi. If the user makes the right choices, Raspi will find the treasure. The code uses `if`/`elif`/`else` statements and functions. See the code files for chapter 5 for examples of the functions.

enter the correct secret name ("Tim") and correct secret password ("raspberrypi"). If the secret name is guessed correctly, then the user has to guess the secret password (see figure 5.9) to see the secret message.

If the password is correct, the user has to enter their favorite color. If the color is red, the program will display the secret message (see figure 5.9).

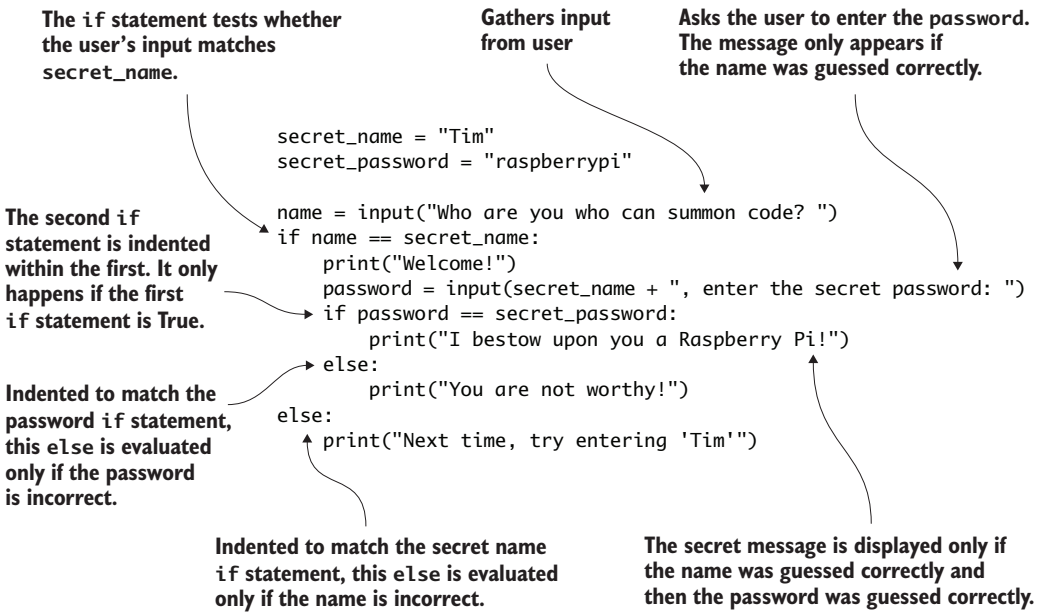


Figure 5.9 `if` statements can be nested within other `if` statements. In this case, the user is only prompted to guess the password if they first guess the secret name correctly. Python uses indentation to figure out what statements belong together and which `if` statements are nested within other ones.

Troubleshooting

A common error when creating `if`/`elif`/`else` statements is forgetting to include the colon at the end of the `if` statement. In this case, when you run the program, you'll see a message pop up in IDLE saying "invalid syntax", and the Python text editor will highlight the end of the line in red (see figure 5.10). You can fix this error by adding a colon at the end of the `if` statement.

Another error is forgetting to put a colon at the end of the `def` statement when creating your own function. In this case, you'll see the same

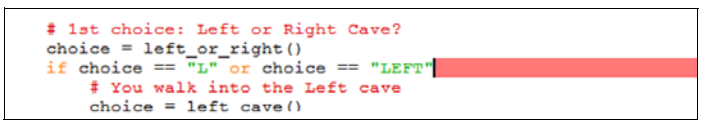


Figure 5.10 Highlighting by IDLE when there is invalid syntax due to a missing colon (`:`) at the end of an `if` statement

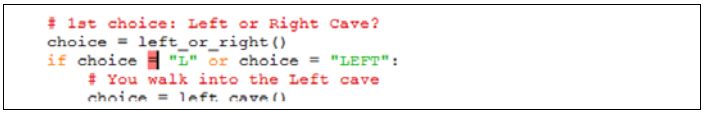


Figure 5.11 Highlighting by IDLE when there is only one equals sign

message ("invalid syntax") and red highlighting at the end of the line missing the colon.

Finally, a third common error is using a single equals sign (=) when comparing two values in an `if` statement. Python will highlight the offending equals sign as shown in figure 5.11. Remember, you need to use a double equals sign (==) to test the equality of two values. This returns True (if the values are equal) or False (if they're not). The single equals sign (=) is used to assign a value to a variable, like `x = 7`.

Fix this error by replacing the single equals sign (=) with double equals signs (==). As you can see, small problems can cause programs to have errors. If you get really stuck, ask a friend to look at your code, or post your code to a forum and ask for help. You'd be surprised by how helpful other programmers are!

Fruit Picker Extra: playing video

In addition to displaying text, as in the cave adventure game, the Raspberry Pi can output sound, show images, and play videos. Let's see how you can play a video on your Raspberry Pi. See appendix A to learn how to set up your Raspberry Pi's Wi-Fi adapter. There are many different video player apps you can use on your Pi, but a great one is OMXPlayer. It was created specifically for the Raspberry Pi and comes preinstalled with Raspbian. We'll explore the audio (or sound) playback capabilities of OMXPlayer in chapter 8.

To show off your Pi's capability, let's play a high-definition demo video from a movie called *Big Buck Bunny*.[2] It's about 10 seconds long and has no sound. Open LXTerminal, and at the prompt enter

```
omxplayer /opt/vc/src/hello_pi/hello_video/test.h264
```

[2] This is a video developed to test video playback and display.

You should see a silent video play for about 10 seconds. Enjoy it! If you know of a video file on the web (.mp4 or H.264 format), OMXPlayer can play it as long as you have a good internet connection. For example, to watch the trailer for another video called *Sintel*, make sure you're connected to the internet and type in

```
omxplayer https://download.blender.org/durian/trailer/sintel_trailer-
➥ 720p.mp4
```

Why not open movies in a web browser? Because OMXPlayer can play them much more easily—it was designed to use the Pi's graphics processing unit (GPU) for playing videos. This means most of your Pi's resources are available to do other things.

Live streaming: exploring from your Pi

You've been pretending to explore a cave. Now let's see if you can use your Pi to explore the ocean or space by live-streaming videos from web cameras. You can turn your Pi into a way to see the sharks and sea turtles by connecting to a live stream coming from the Monterey Bay Aquarium in California. Or maybe you want to see what the Earth looks like from the International Space Station right now.

With a few steps, you can configure your Pi to play live-streaming videos. First you need a small utility called Livestreamer that can take live video streams and output them for OMXPlayer to play, just like your test video. Let's make sure you have the Python package installer. Make sure you have a working internet connection, and then open the Raspbian command line using the Linux Terminal (select Menu-->Accessories-->Terminal), and install the software:

```
sudo apt-get install python-pip
```

After it finishes, install Livestreamer:

```
sudo pip install livestreamer
```

Now you need a link to a live stream of video. Livestreamer will work with many of the most popular live-streaming sites. For this example, you'll use Ustream, but you could also use YouTube Live and many

others. If you go to the Ustream website,[3] you can find links to live-stream videos. Here are few different ones found on the site:

- Watch sharks and turtles at the Monterey Bay Aquarium: www.ustream.tv/channel/9600798.
- Check out the sea life living in the kelp beds at the Monterey Bay Aquarium: www.ustream.tv/channel/9948292.
- See the view from the International Space Station (it may appear dark if the Space Station is in the shadow of the Earth): www.ustream.tv/channel/9408562.

> NOTE These links may change over time. You can get the latest links by searching the Ustream website.

You'll need an internet connection for the next couple steps. You need to figure out the video resolutions available. For the Monterey Bay Aquarium live stream, enter

```
livestreamer http://www.ustream.tv/channel/9600798
```

A few messages appear, and at the bottom are the supported stream resolution(s). For this live stream, you should see a response that says

```
Available streams: mobile_240p (worst, best)
```

This means `mobile_240p` is the only available resolution for the video stream. This is a low-resolution stream, but it's still fun to watch. Tell Livestreamer to send the video to OMXPlayer with this command:

```
livestreamer http://www.ustream.tv/channel/9600798 mobile_240p --
➥ player omxplayer --fifo
```

Great! You should see a video open after a few seconds. It will be low resolution, but sit back and watch the amazing live view of fish, including sharks (see figure 5.12)!

> NOTE Notice that you have to type in `mobile_240p`. You'll type in one of the supported resolutions from the previous step.

[3] Explore the UStream live-streaming videos at www.ustream.tv/explore/all.

Figure 5.12 The Pi's monitor is a live stream from an aquarium. Check out that shark! By using Livestreamer and OMXPlayer, you can stream live video from exotic places, like water holes in Africa and the International Space Station.

Press Ctrl-C to stop Livestreamer and OMXPlayer. Enjoy exploring the world from your Pi!

Challenges

These challenges focus on making improvements to the Raspi's Cave Adventure game. If you get stuck, check appendix C for hints and solutions.

Introducing dramatic pauses

This first challenge is to include some drama in the game by adding two-second pauses between the `print` and `input` statements throughout Raspi's Cave Adventure. This will create anticipation about what will

happen next and give the player more time to read the messages before responding.

Here are some clues for how to accomplish this. First, Python has a built-in `time` module that provides some useful functions for working with time. At the top of the program, you need to add an `import` statement to use this built-in Python toolbox:

```
import time
```

Once you've imported the `time` module, you can call the `sleep` function in the program:

```
time.sleep(1)
```

This example code makes the program pause for 1 second. It takes the form `time.sleep(seconds)`, where `seconds` is the number of seconds you want the program to pause. For example, if you wanted to display a message, wait 3 seconds, and then display another message, you'd write

```
print("It was a dark, dark cave...")
time.sleep(3)
print("Suddenly, a dragon appears out of the shadows.")
```

Go ahead and try to create some drama. If you get stuck, check appendix C or review the code files.

Random demise

Games are always more interesting when they have an element of unpredictability. Try to add some surprises to your game by improving the `wrong_answer` function to randomly display a message from a set of possible ways your player could meet their demise. Here are a couple of examples to get you started:

- Raspi sees a rock on the ground and picks it up. He feels a sharp pinch and drops the rock. He realizes it wasn't a rock but a poisonous spider as he collapses to the ground.
- Standing in the cave, Raspi sees a small rabbit approach. Raspi gets a bad feeling about this rabbit. Suddenly the rabbit attacks him, biting his neck.

Hint: Create `if`/`elif`/`else` statements with different endings, and then use the `random` module to select from the possible endings.

Play again?

Modify the game so that no matter how it ends, the user is always given the option to play again. Hint: Create a variable `play_again` that is initially set to "Y". You'll also need to add a `while` loop to your game that will make the game repeat as long as `play_again` is equal to "Y".

Scream!

If you have a set of headphones or your Pi is connected to a TV with built-in speakers via an HDMI cable, you should be able to play sounds and hear them. Let's look at a simple program to play a sound on your Pi:

```
import os
scream_file_path =
➥ "/usr/share/scratch/Media/Sounds/Human/Scream-male2.mp3"
os.system("omxplayer " + scream_file_path)
```

Test the program, and you should hear a scream. Now see how you can integrate the scream or other sounds into Raspi's Cave Adventure. You can find more sounds on your Pi in the Scratch folder: /usr/share/scratch/Media/Sounds/.

> NOTE OMXPlayer works best with sound files ending in .mp3. Only some files ending in .wav will work. We'll talk more about sound files and the OMXPlayer in chapter 8.

See appendix C if you need help solving these! Good luck!

Summary

You can create engaging programs by putting logic and instructions together into more complex programs:

- Use flow diagrams to map out complex programs before you begin.
- Create flexible programs that can handle unexpected input through the use of Boolean operators.

- Build programs with multiple choices and outcomes using `if`, `elif`, and `else` statements. Chain together multiple `elif` statements to create as many choices as you need.
- When you have logic embedded within logic, nest `if` statements to create decisions that depend on prior choices or conditions.
- Organize your code and cut down on repetition by defining your own functions and then calling them in your program.

Part 3

Pi and Python projects

Let's face it. Pressing buttons, playing sounds, and lighting up cool colored lights is fun! Now you get to use your Pi to make those things happen. You're going to create interactive projects that use your Pi's input and output pins. This makes your Pi a special type of computer that doesn't just show images on the screen, but that can control and sense the world around it. This realm is called *physical computing*. Robotics is physical computing, but think about all the creative possibilities such as making interactive art, creating smart rooms that sense your presence and turn on a light or play music, or producing something that can alert you if it's about to start raining or your pet is drinking water.

In part 3, you'll build projects that can interact with the world using Python and your Raspberry Pi. The projects will require some additional parts that you can purchase individually or as part of a kit, such as the CanaKit Ultimate Kit, Adafruit Starter Kit, or MCM Electronics Starter Kit:

- Raspberry Pi 2 Model B including SD card, power supply, cables, keyboard, and monitor
- Breadboard
- GPIO ribbon cable for the Model B+ (40 pin)
- GPIO breakout board
- 1 dozen jumper wires, male to male

- 1 red LED (light-emitting diode)
- 1 green LED
- 1 blue LED
- 3 push buttons
- 3 resistors, 10K ohm
- 3 resistors, 180 ohm (or between 100 and 300 ohms)
- Headphones or powered computer speakers

You start in chapter 6 by setting up your Pi with an electronics breadboard, building a simple circuit, and controlling an LED (light) using Python. You'll learn how to communicate through your Pi's output pins to make something happen. In this case, you'll make an LED light up. Chapter 7 dives into creating an interactive guessing game that uses lights to respond to a player's input, letting them know with different colors whether their answer is right or wrong. In chapter 8, you'll learn how to listen to your Pi's input pins by wiring up a push button on your breadboard and then responding when it's pushed; and you'll complete a project that combines buttons and sounds to make your own DJ Raspi sound mixer. By the end, the goal is for you to have the knowledge, skills, and confidence to think up and create your own Pi and Python projects.

6

Blinky Pi

In this chapter, you'll be learning about

- *Giving your Pi the ability to talk to the outside world through connectors to anything*
- *Programming the world outside your Pi with simple electric/electronic circuits*
- *Programming the connectors using your previous Python knowledge to make light patterns*

Setting robots in motion, creating smart homes with sensors, and designing an interactive electronic art exhibit sound like vastly different topics, but they're all things you can do with your Raspberry Pi. In each case, the Pi can act as the brain and interact with the world by doing things like

- Checking a robot's sensors and controlling its motors
- Sensing a room's occupants and adjusting the thermostat or lights
- Controlling sound, motion, and light as part of an art display

In this chapter, you'll set up your Pi to control small light bulbs called *light-emitting diodes (LEDs)*. You'll make the LEDs blink using Python. To do this, you'll need to learn a bit about how to build electrical circuits on breadboards. If you've never heard of a breadboard, don't worry! It's

a small board with lots of holes in it to make it easier to build electrical circuits. You'll also be using short wires (called *jumper wires*) to connect certain holes. You'll even learn how to add resistors that keep your LEDs from burning out. See figure 6.1 for a list of parts and what they look like; gather the parts, and let's get started!

Figure 6.1 The Blinky Pi project requires parts that are commonly found in Raspberry Pi starter kits or that can be purchased online.

Setting up your Pi for physical computing

Your Pi is unique compared to most computers because of its input and output pins, called *GPIO* pins. Let's learn how to work with those pins.

> DEFINITION GPIO stands for *general purpose input and output*. These are the pins on your Raspberry Pi that allow it to sense and control things around it.

GPIO pins

The Raspberry Pi 2 Model B and Raspberry Pi 1 Model B+ have 40 pins located on the edge of the board, arranged in 2 rows of 20 pins each (see figure 6.2). Most of the pins on a Pi are used for input and output, so they're often referred to as the Pi's GPIO pins.

> WARNING This project is written for Raspberry Pi 2 Model B. Earlier models of the Raspberry Pi have only 26 pins. See appendix B for information about the differences from the more modern Pi boards. To complete this project with a Raspberry Pi 1 Model B, you may select different pins to light up your LEDs.

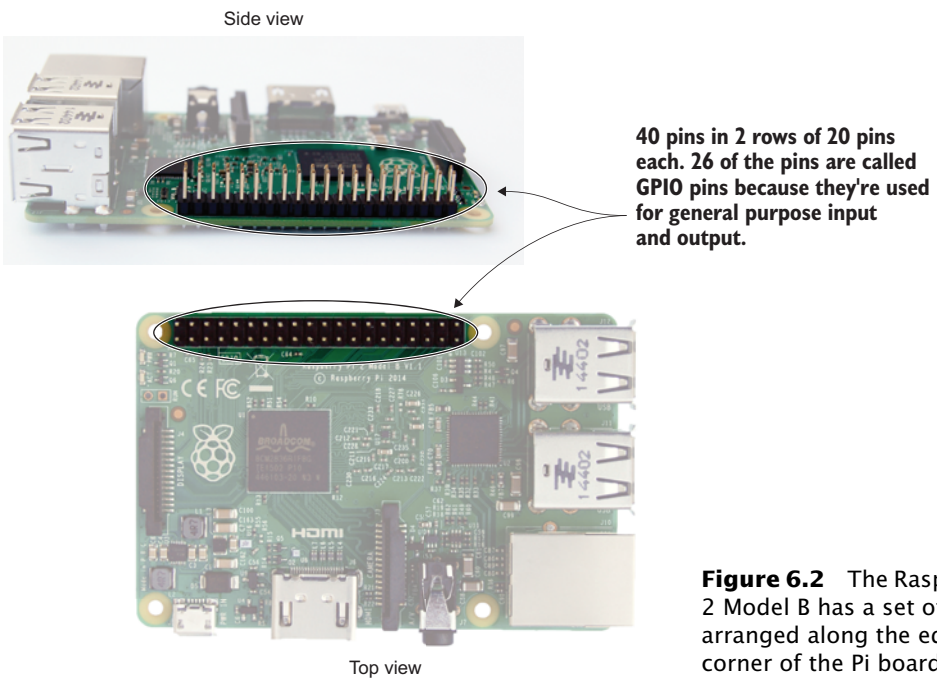


Figure 6.2 The Raspberry Pi 2 Model B has a set of pins arranged along the edge and corner of the Pi board.

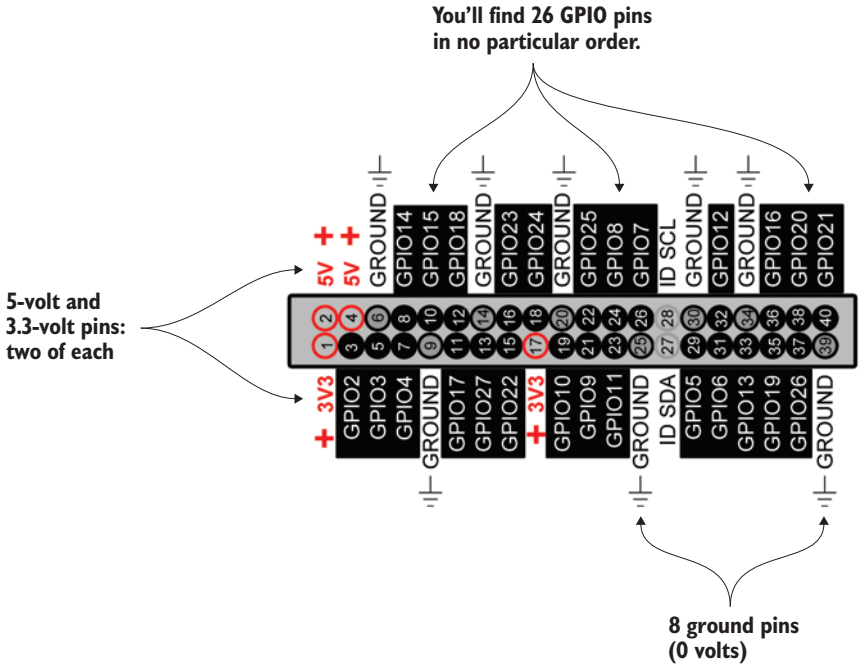


Figure 6.3 The Raspberry Pi B+ has 40 pins. They do different things: some provide 5 volts or 3.3 volts, some are ground pins (0 volts), and many of them are input and output pins that you can program.

Because all the pins look identical, you need a key or diagram to tell you what each one does. Figure 6.3 shows the pins labeled.

Physical pins vs. GPIO pin numbers

In this book, we'll always refer to the GPIO pin numbers, *not* the physical pin locations. The physical pins are numbered from 1 to 40 (shown in the circles in figure 6.3). The GPIO pin numbers go from 1 to 26, and those numbers don't match the physical pin numbers. For example, GPIO 24 corresponds to physical pin 18. By always using the GPIO numbering, it will be easier to wire your circuits and create programs.

Wow, that's a lot of pins! Some pins are for power and are labeled either 3V3 or 5V. These produce 3.3 volts or 5 volts, respectively.

There are also 8 ground pins and 26 GPIO pins[1]—26 pins, just like there are 26 letters in the alphabet.

The GPIO pins support sending out electrical signals (output) or listening for electrical signals from sensors (input). In your body, your brain can send signals to your hand to smack yourself on the forehead (try it!)—this is just like the output from a Pi. Signals are sent out of your Pi to make something happen in the world.

The opposite of output is input. When someone pokes you, your body can detect that poke using nerves in your body. An electrical signal (input) is sent to your brain so you know you've been poked. This is like the way your Pi can be used to detect input or actions in the world.

You'll learn how to output signals in this chapter and chapter 7. Chapter 8 will cover detecting input from the world, such as detecting when a button has been pressed.

Let's get ready to connect some wires! But wait: connecting an LED directly to the GPIO pins on the board of your Raspberry Pi isn't feasible, because the pins are so close together. What can you do? You need more space to build circuits.

Breaking out the GPIO pins to a breadboard

To give you room, you'll move the GPIO pins over to a breadboard. This is called *breaking them out*. To do this, you need a ribbon cable, breakout board, and solderless breadboard (see figure 6.4).

Breadboards make it simple to prototype circuits. Like a park might provide large, open fields that make it easy to play sports, think of a breadboard as a nice, open electrical playing field where you can play with electrical parts. The breadboard allows you to plug wires and components into small holes. You can build and rebuild circuits on a breadboard with little effort.

[1] Oddly, you'll notice that the GPIO pins are numbered from 2 to 27. Pins 0 and 1 are used for communicating with other computer chips using a super-special protocol called I2C. These are labeled ID SDA and ID SCL in figure 6.3.

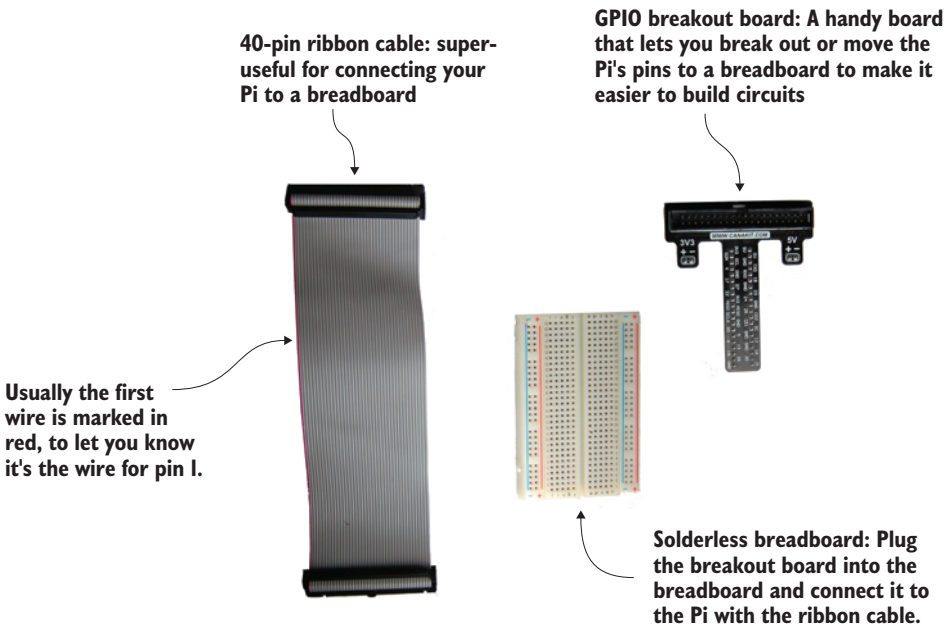


Figure 6.4 To easily create projects using your Pi's GPIO pins, you can connect the Pi to a breadboard using a ribbon cable and breakout board. The parts shown are examples of the ones commonly found in many Raspberry Pi kits.

Find your breakout board, and insert it into the top of the breadboard. Line up the pins before you push it down *hard* (see figure 6.5). Your particular breakout board may look a little different, but they all act the same. With the breakout board in place, it'll be easier to build circuits with your GPIO pins.

Connect one end of the ribbon cable to the Pi's GPIO pins; line it up carefully before you push it down. Then connect the other end of the cable to the breakout board on your breadboard (see figure 6.6). A breakout board has a notch in it so the ribbon cable will only fit one way.

> WARNING Ribbon cables usually have a stripe that marks the first wire. White or grey ribbon cables often use a red stripe. Black ribbon cables often have a white stripe. These mark the first wire on the cable. Make sure this first wire is connected toward the edge of your Pi's board and away from the USB ports.

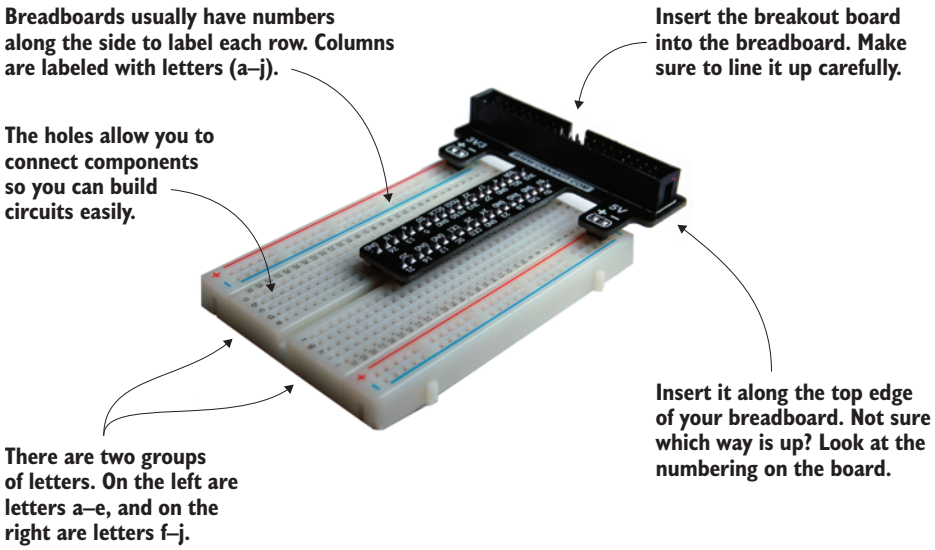


Figure 6.5 Carefully line up the breakout board, and then press it firmly into the breadboard. The two rows of pins on the breakout board should straddle the center gap.

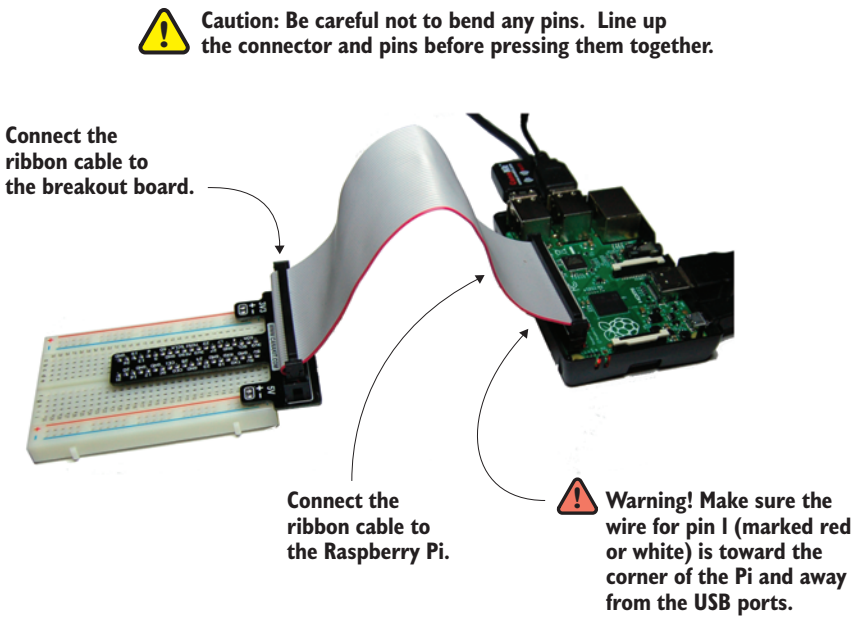


Figure 6.6 Connect one end of the ribbon cable to the breakout board. Connect the other end to your Raspberry Pi.

Breadboard basics

A breadboard[2] has a set of internal connections that you can't see. But if you had X-ray vision, you'd see that certain holes are connected. Let's look at the connections in your breadboard (see figure 6.7).

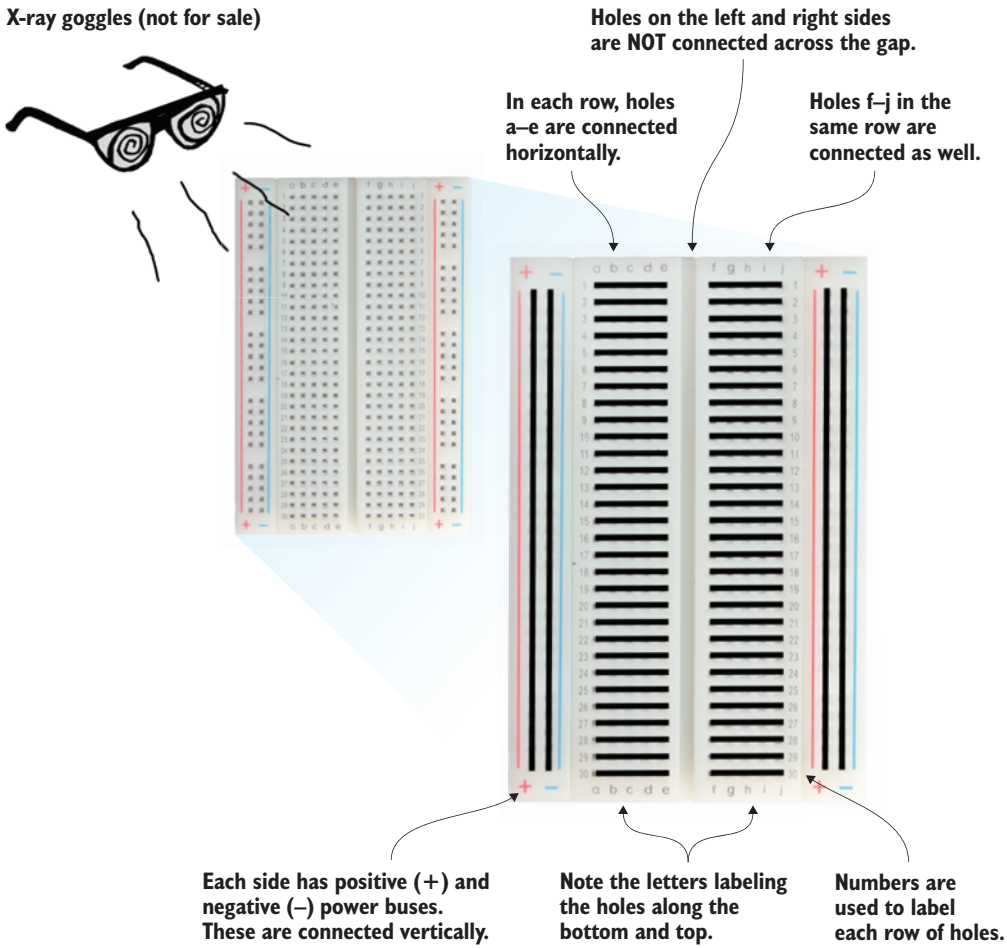


Figure 6.7 Breadboards have internal connections. You need to know about them in order to build circuits. Rows of pins are connected horizontally, but not across the gap in the middle. Long rails called *power buses* run vertically along the sides of the board.

[2] Prior to the development of the kind of breadboards we're using, people built circuits on pieces of wood that were used to cut bread on (hence the name). They needed a quick way to connect circuits, and by drilling holes and using nails and wires, they could use bread boards to try different circuits.

On this breadboard, rows are labeled with numbers (1–30), and the columns have letters (a–e on the left side and f–j on the right side). You can refer to a specific hole in the breadboard by saying its row number and letter. For example, if you wanted to refer to the hole located in row 25, column c, you could say 25c (see figure 6.8). Just as you might find your seat at a stadium by walking along the aisle to find the correct row, and then moving along the row to find the right seat, you'll use the letters and numbers to guide you in building your circuits.

BREADBOARD (BB) HOLES

We'll refer to the row and column, but we'll prepend the letters *BB* so you know it's the breadboard location we're talking about. Figure 6.8 shows the location of BB25c. If we're talking about a GPIO pin or connection, we'll add *GP* before the number (GPIO pin 21 is GP21).

Try to keep in mind what is connected in a breadboard and what isn't. If you forget, you can always look back at figure 6.7. For example, notice that BB25 a, b, c, d, and e are all connected. Similarly, BB30 f, g,

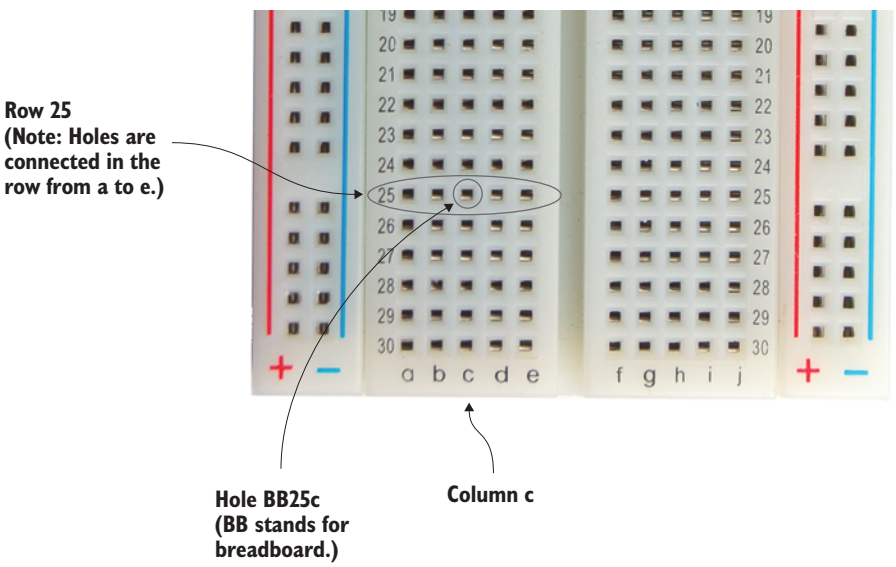


Figure 6.8 To find a specific hole on a breadboard, use the row and column labels. This is a close-up of a breadboard, showing how you can find the location of hole 25c (we'll refer to the hole as BB25c, where *BB* stands for *breadboard*).

h, i, and j are connected. But the left side of the board isn't connected to the right. For example, BB25e isn't connected to BB25f. To connect them, you'd put a jumper from BB25e to BB25f.

You can see vertical columns of holes along the sides of the breadboard. These are the *power buses* and provide easy ways to connect electrical components to power (positive) and ground (negative).

Circuits 101

Let's learn about electricity and circuits. At the simplest level, a *circuit* is a loop or path where the electrical power starts at a source (the positive side of a power source), goes through one or more electrical components (such as a light or motor), and then completes the loop (or path) by connecting back to the negative side of the source.

What is electricity?

Electricity is the flow of charge. Typically, it is the flow of electrons, which have a negative charge. To get electrons to flow, you need to have a difference in charges. Just as the north pole of a magnet is attracted to its opposite—the south pole of another magnet—positive and negative electric charges are attracted to one another. If the charge is free to move, it will move. We generally think of circuits as having electricity flowing from the positive (+) side of the source to the negative (-) side of the source. For your Pi, the power is coming from the power supply (Micro USB plug). The Pi as a power source can provide either +3.3 volts or +5 V (volts). It provides this power through the physical pins 1, 2, and 4, but can also send +3.3 V out any of the 26 GPIO pins (you'll program it to do that soon).

Voltage (volts)

Voltage is a measure of the difference in electrical charge between the positive and negative source. When you have two different charges, they're attracted to one another (positive and negative attract). The greater the difference in charge, the greater the force (or electrical pressure) wanting to move charges through the circuit from the positive side to the negative side.

Voltage is measured in volts (V), named after Alessandro Volta, who is credited with inventing the first battery. A 9 volt (or 9 V) battery has a greater electric force for moving charge than a AA battery, which only has a voltage of 1.5 V.

Current (amperes)

The *current* in a circuit is the amount of charge flowing. So whereas voltage is a measure of how badly charges *want* to flow, the current is a measure of how much charge is *actually* flowing.

Imagine that you could be inside a wire and see the charge flowing through it. A large current would mean a lot of charge (usually electrons) bumping along and

through the wire over some period of time. A small current in that same wire would mean a lot less charge flowing over that same time period. Current is measured in amperes (A), named after André-Marie Ampère. A current of 1 ampere (or 1 A) is equivalent to the amount of charge of 6.241×10^{18} electrons flowing through a wire per second! That is a lot of charge flowing. You can decrease the current in a circuit by increasing the resistance of the circuit to the flow of electric charge.

Resistance (ohms)

The *resistance* in a circuit is a measure of how much it opposes the flow of charge (current). A light bulb, a motor, and your body all have resistance. The opposite of resistance is *conductance*. Substances such as metal (copper, silver, and gold) are all good conductors, and this is why we build circuits with metal wires for the electricity to flow through.

Sometimes you need to control the current (the flow of charge). Resistors are used to do this; they're made of materials that slow down the flow of charge. The most common ones are made out of carbon (you'll be using these in your projects). The resistance of a circuit is measured in ohms, named after Georg Ohm, and is represented using the Greek symbol omega (Ω).

Pi circuits

You can think of your Pi as providing 3.3 V from the positive side of the Pi or, later, coming out of one of the GPIO pins. This +3.3 V is a force that is trying to push electric charge to the negative (-) side of your source. The negative side is sometimes called the *ground*—think of it as a big sink or reservoir to which electricity wants to flow if there is a path to get there. During the next few chapters, you'll build circuits with LEDs and resistors. You use a resistor with an LED to decrease the flow of electric charge (the current) so it won't be too large and burn out your LED. Burning an LED smells bad!

On your breadboard, think of all the GPIO pins as potential sources of voltage (positive). Circuits from the GPIO pins should end back at any one of the many ground (negative) connections.

Building the LED circuit

Your first project is to light up a red LED. You'll control the LED using GPIO pin 21 (GPIO21). You need these parts:

- Raspberry Pi, ribbon cable, and breakout board connected to your breadboard
- 1 red LED (5 mm)
- 1 180 ohm resistor
- 1 jumper wire (male-to-male)

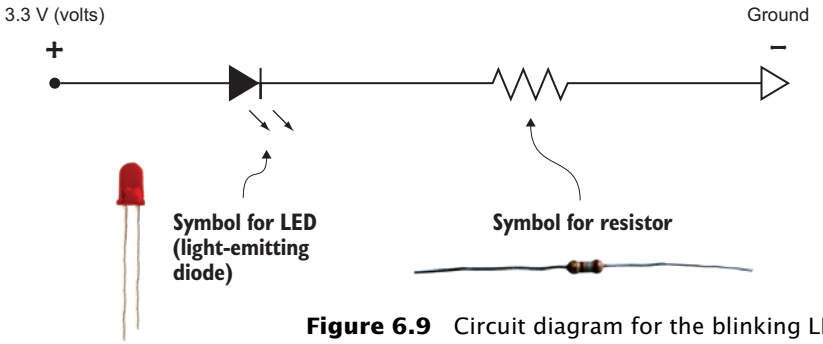


Figure 6.9 Circuit diagram for the blinking LED project

You'll build the LED circuit on your breadboard and then program it to light up. Figure 6.9 shows the circuit diagram. To light the LED, you'll have electricity (+3.3 V) flow from your Pi's GPIO pin 21 through the LED, through the resistor, and then to ground (0 V).

Figure 6.10 shows the LED circuit built on the breadboard. Note that there are many different ways to create this circuit—this is just one way. Let's walk through the steps to build the circuit.

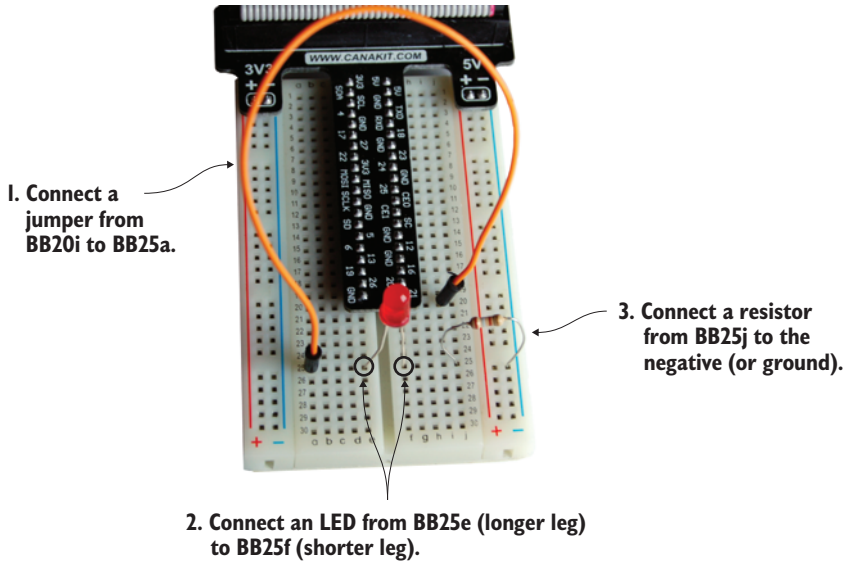


Figure 6.10 LED circuit built on the breadboard. You're using GPIO pin 21 as the power source. The light won't turn on until you program the voltage to come out of the pin.

NOTE You may have a different breadboard than the one used in this book. If so, the numbering on your breadboard may be different than what is shown here. In that case, you'll need to create the circuit following the same principles, but with different numbered holes.

Step 1. Connect the jumper from GPIO pin 21

Raspberry Pi GPIO pins can output 3.3 V. You could pick any pin, but this project uses GPIO pin 21.

Connect a short piece of wire from GPIO21 on your breadboard to an empty row on the breadboard. Use row 25. Firmly push the wire into the hole. The metal tip of the wire should go down into the hole, not sit on top.

The breakout board pins are connected to rows on the breadboard. We'll refer to the holes on the breadboard (see figure 6.11). Insert one end of the jumper into BB20i and the other end into BB25a.

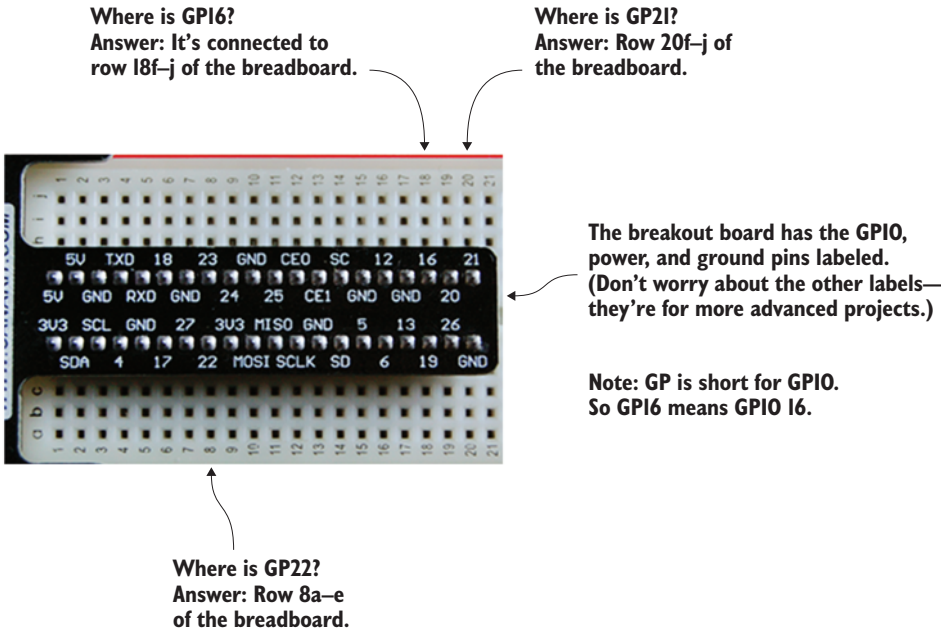


Figure 6.11 The breakout board has labels that correspond to the pins on your Pi. To connect a wire to GP16, you plug it into the breadboard in the hole labelled BB18f or BB18j.

Step 2. Add the red LED

It's time to connect the red LED. LEDs only let electricity flow through them one way, so it's important to put them in the right way. LEDs have two wires or *legs*. The longer leg is called the *anode* and connects to the positive side of the circuit (see figure 6.12). The shorter leg, called the *cathode*, connects to the negative or ground side of the circuit.

With the red LED, connect the longer leg to BB25e and the shorter leg to BB25f. You may need to bend the legs and push them a bit to get them into the holes.

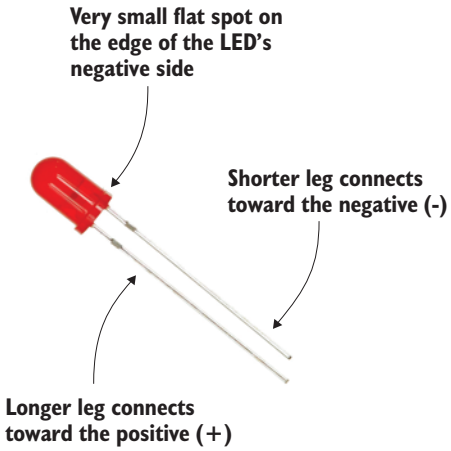


Figure 6.12 LEDs have two legs (wires) coming out of them. The longer leg is called the anode and connects to the positive side of the circuit. The shorter one is called the cathode and connects to the negative side of a circuit.

Step 3. Connect a resistor

Grab your 180 ohm resistor.[3] You can identify a resistor by its color-coded bands. A 180 ohm resistor has colored bands of brown, grey, and brown (see figure 6.13). They are followed by a fourth band that

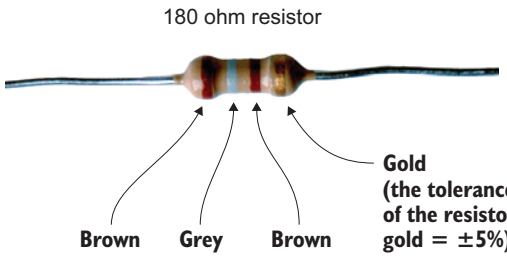


Figure 6.13 The value of a resistor is determined by its colored bands. See the sidebar "Resistor color codes" for a chart; there are also many online color-code charts.

[3] If you don't have a 180 ohm resistor, you can use a resistor with a value between 100 and 330 ohms. If you use a resistor with a value that is too large, the LED may not light up or will be dim. Try experimenting with different resistors to adjust the brightness.

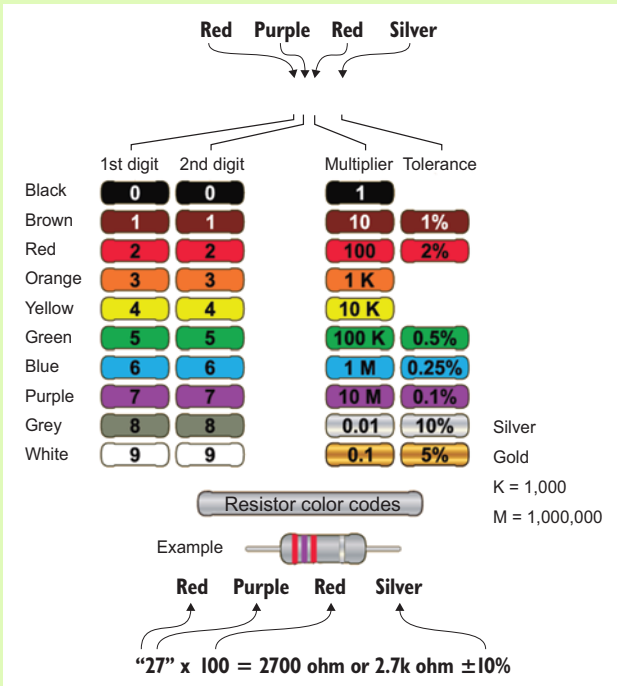
indicates the tolerance or quality of the resistor. Common colors for the fourth band are gold (±5% tolerance) and silver (±10% tolerance).

The resistor prevents too much electric current[4] from passing through your LED and burning it out. Insert one end of the 180 ohm resistor into BB25j and the other end into the negative (-) power bus (or ground).

Electricity will flow either way through a resistor, so which way you connect it doesn't matter. Remember that the negative power bus or ground rail is running vertically along the right side of the breadboard. Most boards have a blue stripe next to it.

Resistor color codes

Resistors have color codes that tell their value and tolerance. This chart shows you how to read the resistor color bands.



[4] Current is a measure of the flow of electric charges per second. If the current through an LED is too high, the LED will burn out.

(continued)

For example, consider a resistor with red, purple, red, and silver bands. Follow these steps to use the chart:

- Look up the digit for the first band and the digit for the second band, and put them together. In this case, the digits are 2 and 7: put them together, and you get 27. Note that you don't add the numbers; you treat them as the first and second digits of the resistor value.
- Find the multiplier by looking up the color for the third band. In this case, it's 100 ohms (red).
- Put it all together: 27 × 100 ohms is 2,700 ohms or 2.7K ohms (K = 1,000).
- The fourth band (silver) tells you the resistor has a tolerance of ±10%.

A red, purple, red, and silver resistor is a 2.7K ohm resistor with a ±10% tolerance. Use this handy chart any time you need to look up the value of a resistor.

That's it! You have a completed LED circuit built on your breadboard. Now it's time to program it!

Software: blinkLED program

Open IDLE by choosing Python 3 under Menu > Programming. This opens IDLE to the Python 3.x Shell. In the Python Shell, let's check to see if your Pi has the GPIO libraries you need already installed:

```
>>>  import RPi.GPIO as GPIO
```

If you don't see an error, you're ready to go. If you see an error saying there is no module named `RPi.GPIO`, please refer to the sidebar "Updating your Pi."

Updating your Pi

Before programming, you need to check that your Pi is up to date. Make sure your Pi is connected to the internet. Open the Terminal program by going to Menu --> Accessories --> Terminal, and run the following commands to update your Raspberry Pi and be certain you have the Raspberry Pi GPIO packages you need.

First, let's update the `apt-get` database. The `apt-get` program handles installing and removing software from your Pi. In Terminal, enter this command:

```
pi@raspberrypi ~ $ sudo apt-get update
```

You'll need to wait while a bunch of files are downloaded and installed. You'll see lots of messages displayed in Terminal. When the command completes, you'll see the Terminal $ prompt again. Next, to get the latest Pi software, enter

```
pi@raspberrypi ~ $ sudo apt-get upgrade
```

Once again, files will be downloaded and installed. After a series of messages, you'll see a warning about the upgrade using additional disk space, and this prompt: "Do you want to continue [Y/n]?" Enter Y and press Enter to continue the upgrade.

This is a great time to grab a sandwich and soda. It can take 15 minutes or more for the update to complete. When it's finished, you'll have the latest Raspberry Pi software and Python libraries, including the ones you need to communicate with and control the GPIO pins.

You're going to write a program that blinks an LED. It'll send a voltage (+3.3 V) out of a GPIO pin to light the LED, then turn it off, and repeat that over and over. Begin by creating the following new program in IDLE. In the Python Shell, start a new program by pressing Ctrl-N or selecting File > New Window.

Listing 6.1 Blinking LED program

```
import RPi.GPIO as GPIO
import time

# Variable for the GPIO pin number
LED_pin_red = 21

# Tell the Pi we are using the breakout board pin numbering
GPIO.setmode(GPIO.BCM)

# Set up the GPIO pin for output
GPIO.setup(LED_pin_red, GPIO.OUT)

# Loop to blink our led
while True:
    GPIO.output(LED_pin_red, GPIO.HIGH)
    print("On")
    time.sleep(1)
    GPIO.output(LED_pin_red, GPIO.LOW)
    print("Off")
    time.sleep(1)
```

Load the libraries you need to control the GPIO pins.

Create a variable for the GPIO pin number you're using to control the LED.

Set up GPIO pin 21 as an output.

Turn on GPIO pin 21 (GPIO.HIGH means turn on).

Turn off GPIO pin 21 (GPIO.LOW means turn off).

Save the program as blinkLED.py in your home folder. The program can't be run the same ways you've run programs before using IDLE.

Running the program

Select Run > Run Module (or press F5) from the IDLE text editor to run your program.With older versions of Raspbian, programs using GPIO pins must be run from the Raspbian command prompt as the superuser (or root)[5]. If you run the program at the Python Shell in IDLE, you'll get an error:

```
RuntimeError: No access to /dev/mem. Try running as root!
```

In this case, you use the `sudo` command to do this. To run the blinkLED.py program, open LXTerminal and enter the following command:

```
pi@raspberrypi ~ $ sudo python3 blinkLED.py
```

Behold the blinking LED! Try making the light blink faster by adjusting the value in the `sleep` function. Use a smaller number of seconds, such as 0.5 or 0.1.[6] To stop the program, press Ctrl-C.

> NOTE Stopping the program with Ctrl-C may result in the light being left on (depending on when you press it). Also, the next time you run the program, you may see a runtime error, but the program still works. We don't cover it here, but look online for the Python commands `try/except/finally` and the `GPIO.cleanup()` command. It's a fancy way to make sure all the GPIO pins are reset when you exit the program.

TROUBLESHOOTING

If the light isn't blinking, here are some things you can check:

- Are the on and off messages displaying on the screen? If so, it's probably not your code that has a problem. Check the circuit on the breadboard. Make sure the ribbon cable is connected properly, with

[5] In October 2015, the Raspberry Pi Foundation released Raspbian version "Jessie," which allows you to run programs using the GPIO pins directly from IDLE. With "Jessie" you don't need to open the command prompt. Simply press F5 or select Run > Run Module from the IDLE text editor menu to run your programs.

[6] Too small a number may cause the light to appear to stay on, but more dimly. This is because your eyes can only perceive blinking that is greater than about 1/25th of a second, or 0.04 of a second.

the first wire connected toward the edge of your Pi, away from the USB ports. Double-check that the jumper, LED, and resistor are connected to the correct holes.

- Could your LED be inserted the wrong way? Make sure the shorter leg is toward the negative or ground side. Try turning it around.
- Double-check the size of the resistor you used in the circuit. If the resistor is too large, the LED won't light up. A resistor that is between 100 and 300 ohms should work.
- Look through your Python program for errors. Check that you have set `LED_pin_red` equal to 21 and that you're setting it `HIGH` and then `LOW`.

blinkLED: how it works

Let's take a closer look at how the blinkLED.py code works.

LOADING LIBRARIES

The `import` commands load the libraries or toolboxes you want to use in your program:

```
import RPi.GPIO as GPIO
import time
```

These commands load the Python libraries for controlling the Pi's GPIO pins. They also load the `time` library so you can use the `sleep` function to control the rate of blinking.

Importing libraries with the as keyword

Notice the `as` keyword in `import RPi.GPIO as GPIO`. Why can't you just type `import RPi.GPIO`?

The `as` keyword tells Python to load the library to a certain name you specify. It's kind of like giving the whole library a nickname. In this case, it's so you can refer to `RPi.GPIO` as simply `GPIO`.

An example will make it clearer. Once you've imported the `RPi.GPIO` library as `GPIO`, you can type `GPIO.setmode(GPIO.BCM)`. Without it, you would have to type `RPi.GPIO.setmode(RPi.GPIO.BCM)`. You can see how using `as GPIO` saves you some typing!

Once the libraries are loaded, you can set up your GPIO pins.

SETTING UP A GPIO PIN FOR OUTPUT

To set up a GPIO pin, you first need to tell Python on your Pi that you'll be referring to pins by the standard breakout numbering scheme. These are the numbers printed on the breakout board. You use the `setmode` function:

```
GPIO.setmode(GPIO.BCM)
```

BCM stands for Broadcom—the maker of the computer chip that the Pi uses. Next you tell your Raspberry Pi that you'll be using `LED_pin_red` (GP21) for output, meaning you're planning to send some electricity out of it:

```
LED_pin_red = 21

GPIO.setup(LED_pin_red, GPIO.OUT)
```

`GPIO.OUT` prepares GP21 to send out +3.3 V of electricity.

LOOPING AND BLINKING

Finally, you create an infinite `while` loop and turn the LED on (set `GPIO.HIGH`) and off (set `GPIO.LOW`). You also add a delay using the `sleep` method found in Python's `time` library. Notice how the `sleep` function takes a parameter that is the number of seconds to sleep or pause. In this case, you use 1 second:

```
while True:
    GPIO.output(LED_pin_red, GPIO.HIGH)
    print("On")
    time.sleep(1)
    GPIO.output(LED_pin_red, GPIO.LOW)
    print("Off")
    time.sleep(1)
```

The `print` commands display messages to the screen. Although they aren't necessary to blink the LED, they can help debug your program. If you do use them, the screen could quickly fill with messages. Set a longer delay time to prevent this. If you see the messages on the screen but your LED isn't lighting up, then you probably have an error in your circuit and not in your program. Check your wiring, try turning around the LED, or try a different LED in case that one is defective.

Adding more LEDs

One LED is fun, so three LEDs must be lots of fun. Let's try adding green and blue LEDs and modify the program to control them. Here are the parts you need:

- Raspberry Pi and circuit from before
- 1 green LED
- 1 blue LED
- 2 180 ohm resistors
- 2 jumper wires (male-to-male)

Building the circuit

You'll follow the same process as before to add the green and blue LEDs. Figure 6.14 shows what the circuit diagram looks like now, and figure 6.15 shows the circuit on a breadboard.

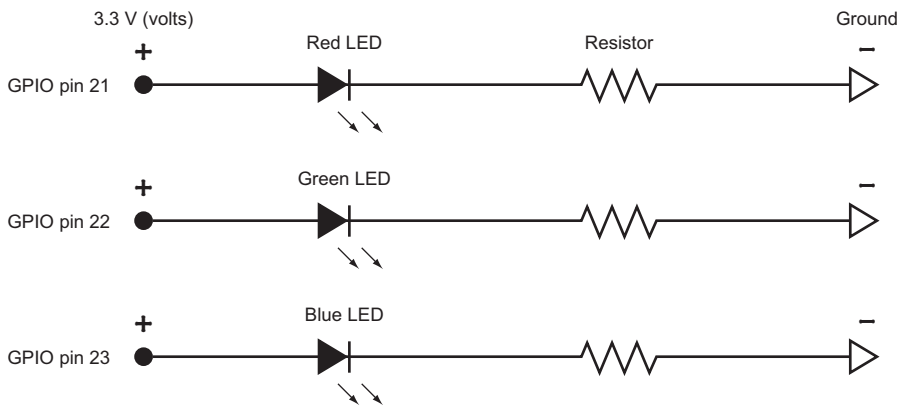


Figure 6.14 Circuit diagram for three LEDs: red, green, and blue. You'll use 180 ohm resistors like before. They will all be controlled by different GPIO pins. Red will use 21, green will use 22, and blue will be connected to pin 23. You could use any of the 26 different GPIO pins.

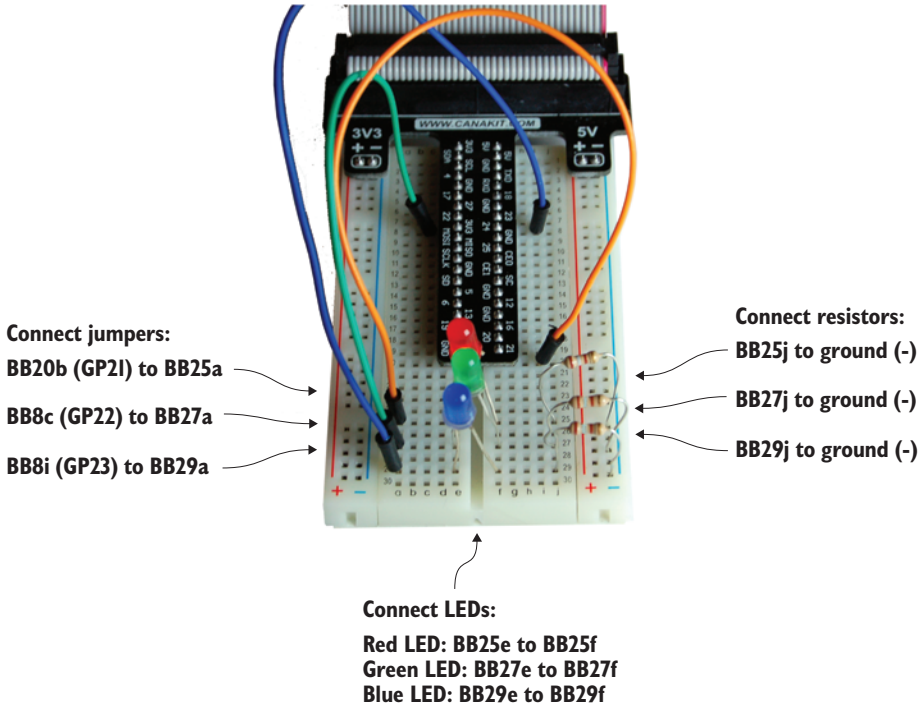


Figure 6.15 The three-LED circuit is built on the breadboard. Each LED and its corresponding resistor are placed in a row together. This example uses rows 25, 27, and 29.

To add the green LED, follow these steps:

1. GP22 is located on the *left side of the breakout board in row 8* on the breadboard. Connect it to *row 27*: insert one end of the jumper into BB8c and the other end into BB27a.
2. Connect the long leg of the green LED to BB27e and the shorter leg to BB27f. Bend the legs if needed.
3. Connect a 180 ohm resistor (brown, grey, and brown) from BB27j to the closest hole in the negative power bus.

Here are the steps to add the blue LED:

1. GPIO23 is located on the *right side of the breakout board in row 8* on the breadboard. Connect it to *row 29*: insert one end of the jumper into BB8i and the other end into BB29a.

2 Connect the long leg of the blue LED to BB29e and the shorter leg to BB29f. Bend the legs if needed.

3 Grab a 180 ohm resistor. You guessed it! It's color-coded brown, grey, and brown. Connect it from BB29j to the closest hole in the negative power bus.

Multiple LEDs: program it!

You need to make a few changes to the program to add more LEDs and get them all blinking at the same time. The following listing shows the updated code.

Listing 6.2 Three blinking LEDs

```
import RPi.GPIO as GPIO
import time

# Variable for the GPIO pin number
LED_pin_red = 21
LED_pin_green = 22
LED_pin_blue = 23

# Tell the Pi we are using the breakout board pin numbering
GPIO.setmode(GPIO.BCM)

# Set up the GPIO pins for output
GPIO.setup(LED_pin_red, GPIO.OUT)
GPIO.setup(LED_pin_green, GPIO.OUT)
GPIO.setup(LED_pin_blue, GPIO.OUT)

# Loop to blink our LEDs
while True:
    GPIO.output(LED_pin_red, GPIO.HIGH)
    GPIO.output(LED_pin_green, GPIO.HIGH)
    GPIO.output(LED_pin_blue, GPIO.HIGH)
    print("On")
    time.sleep(1)
    GPIO.output(LED_pin_red, GPIO.LOW)
    GPIO.output(LED_pin_green, GPIO.LOW)
    GPIO.output(LED_pin_blue, GPIO.LOW)
    print("Off")
    time.sleep(1)
```

Create variables for the GPIO pins you're using for the green and blue LEDs.

Set up GPIO pins 22 and 23 as outputs.

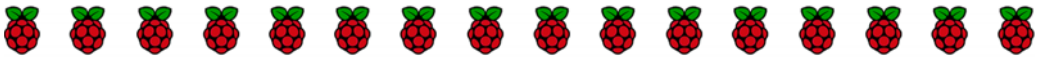
Turn on the GPIO pins.

Turn off the GPIO pins.

Save the code as blinkLED3.py, and try running it. Open LXTerminal, and enter the following command:

```
pi@raspberrypi ~ $ sudo python3 blinkLED3.py
```

Fantastic! You have your own light show going on!



Challenges

Try these challenges to practice controlling your Raspberry Pi's GPIO pins. Each one provides a unique problem to solve.

Wave pattern

Change the program to make each LED turn on, one at a time, until they're all on. Then, turn each LED off, one at a time. Hint: play with where you put the `time.sleep(1)` command. Can you make the LEDs light up and turn off in a wave pattern?

Simon Says

Write a function that blinks the LEDs and that can take five parameters representing a pattern of colorful blinks. Each parameter is a string representing a color: red, blue, or green. The function should blink the lights in the appropriate pattern. Here is a series of Simon Says patterns you should try to make your function produce:

Red, green, red, red, blue

Blue, green, blue, green, red

Green, blue, blue, red, green

Random blinking

Create a program that generates random durations for how long the lights stay on and off. The durations should be random floating-point numbers between 0 and 3 seconds. Hint: you can use the `random` method

to generate a random floating-point number between 0 and 1.0. Here is an example:

```
off_random_time = random.random() * 3
```

To scale this number so that it's between 0 and 3, you can multiply `off_random_time` by 3. If you get stuck on the challenge, check appendix C and the chapter source code for hints and solutions.

Summary

In this chapter, you learned the following things:

- A Pi is capable of interacting with the world around it. With a few extra parts, you can set it up for physical computing projects.
- A Pi can send out electrical signals! You can send output through the GPIO pins, and this can be used to light up LEDs or control many other electronic components (motors, buzzers, relays, and so on).
- Breadboards are like playgrounds for electronics. They make it easy to create circuits for your Pi because you can easily build and take apart circuits for use with the Pi.
- The `RPi.GPIO` library has built-in functions to set up and control output (voltage) to GPIO pins with Python.

Just imagine the possibilities of controlling pretty much any electrical device using your Raspberry Pi. Even better, imagine making the device work based on sensors (inputs) so you can create smart devices programmed by you!

7

Light Up Guessing Game

In this chapter, you'll be learning about

- *Simplifying and improving your code with more thoughtful design and use of functions*
- *Building a circuit to control a special LED (light bulb) that can make and combine red, green, and blue light*
- *Adding together colors of light to create new colors*
- *Making your Pi come alive by having it respond using different colored light*

Your Raspberry Pi has a unique ability to interact with the world around it. In the last chapter, you made lights blink based on a programmed pattern. Nice, but that isn't truly interactive, because the Pi always blinks a pattern that you program it to do. In this chapter, let's see if you can create an interactive project that *responds* to you through its GPIO pins. You'll draw on what you've learned about conditional logic (`if`/`elif`/`else`) to have your Pi make decisions and respond. As you did in earlier chapters, you'll need to gather input, use loops, and apply a few other programming techniques to get it done.

You're making a Light Up Guessing Game, but not just any one: this game will illuminate a small light called an *RGB* (stands for red, green, blue) *LED*, which can make any color. You'll use your Pi, breadboard,

and electrical parts, along with a program you're going to write. Your Pi will let the player know if they're correct by flashing the RGB in different colors if their guess is too high or too low.

Figure 7.1 shows the parts you need. You'll notice that some of them are the same as in chapter 6, but you'll also need an RGB LED. Let's get started!

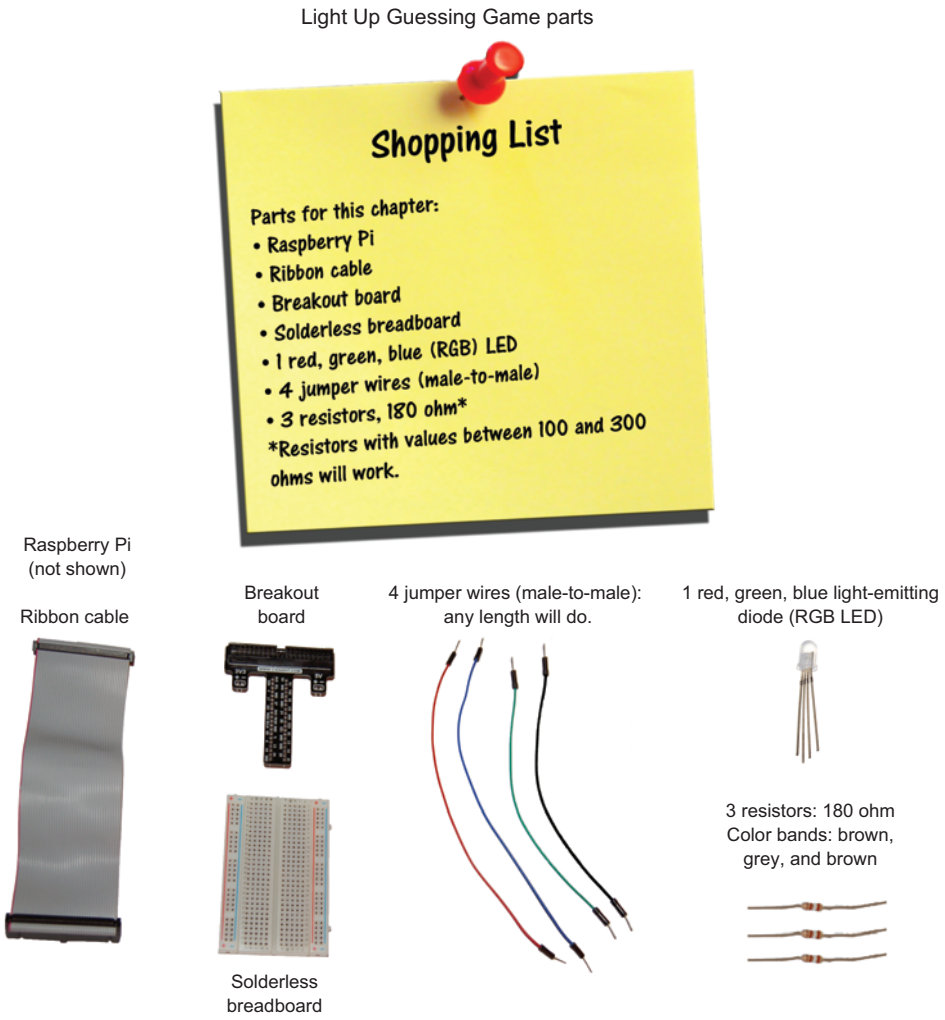


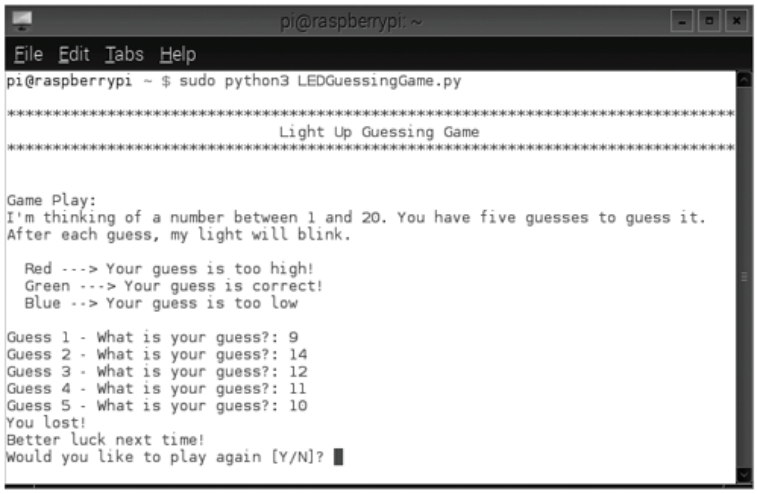
Figure 7.1 The Light Up Guessing Game uses a red, green, blue (RGB) LED. An RGB LED can produce many different colors because it has three LEDs (colored red, green, and blue) packed inside it.

Guessing Game design

The object of the game is to guess a magic number. This time, the Pi will give feedback to the user by lighting up the RGB LED in different colors. Here are some game details:

- The magic number is a randomly generated number between 1 and 20.
- The player is given five tries to guess the number correctly.
- If they guess correctly, the RGB LED flashes green.
- If the guess is too high, the RGB LED flashes red.
- If the guess is too low, the RGB LED flashes blue.
- The player is given the choice to play again.

Figure 7.2 shows a sample of the game's output.



Light flashes different colors to respond to the player's guesses

Figure 7.2 The Light Up Guessing Game responds to the user after each guess. Lights on the breadboard light up to let the player know if their guess is too high or too low.

You'll approach this project in two parts. The first part is to build the circuit (the hardware), and the second part is writing the program (the software).

Hardware: building the circuit

Let's get building! You're building a circuit on your breadboard to control a new type of LED that can make any color you want. You'll start by connecting your Pi's GPIO pins to the breadboard using the ribbon cable and GPIO breakout board. Refer back to chapter 6 (section 6.1) if you need a reminder about how to set this up. Your Pi and breadboard should look like figure 7.3.

Numbers, numbers, numbers!

As first explained in chapter 6, you need a way to find a particular hole on your breadboard, and to do that you'll use the numbers and letters. Remember, this is much like the way you might find your seat at a stadium for a concert or sporting event.

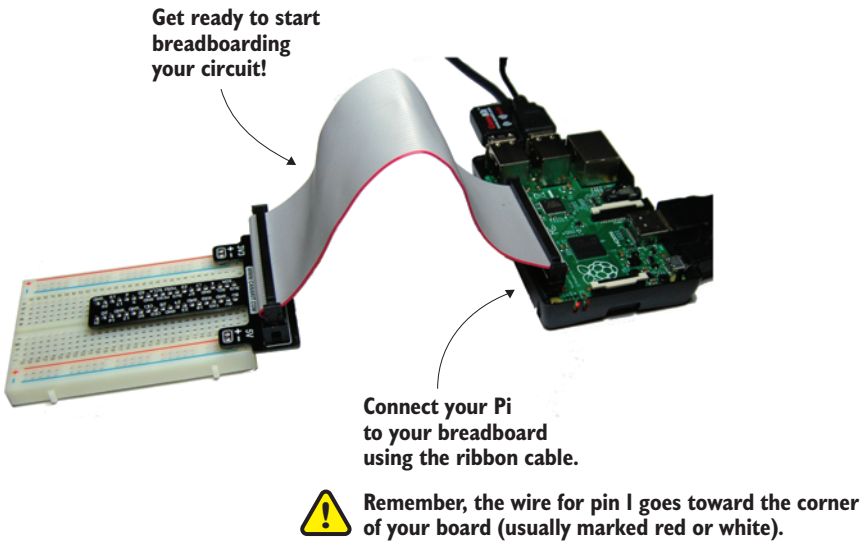


Figure 7.3 The Pi, breakout board, and breadboard setup. And you thought your desk was messy before!

To refer to a specific hole on the breadboard, we'll refer to the row and column, but we'll add the letters *BB* to stand for *breadboard*. Not too hard, right? Finding breadboard holes involves searching for the row and then the column. When referring to a GPIO pin, we'll add the letters *GP* in front. For example, GPIO pin 12 is referred to as GP12.

Wiring an RGB LED

You're wiring up a new type of LED, called an RGB LED.

> DEFINITION An RGB LED is a light bulb that consists of three LEDs: one red (R), one green (G), and one blue (B), all in a single plastic LED bulb casing.

The RGB LED can produce pretty much any color you want, using the three tiny LEDs inside it. By powering these in varying amounts, you can mix light to make colors.

The RGB LED has four *legs* (or wires) coming out of it, so you'll need to figure out how to wire it up. It's a bit different than the single-color LEDs you wired up in chapter 6, but it's pretty easy to use.

Circuit sketch

The circuit diagram for the Light Up Guessing Game is shown in Figure 7.4. To light the RGB LED, you'll have electricity (+3.3 V) flow from your Pi's GPIO pins 12, 16, and 21; through each resistor; through the LED; and then to ground (0 V).

You'll build the RGB LED circuit on the breadboard and then program it to light up. Wire it up in this order:

1. Put the RGB LED into the breadboard.
2. Connect the three jumper wires, which will connect the GPIO pins to the LED (one for each color).
3. Add the three resistors to connect the jumpers to the LED's red, green, and blue legs.
4. Add the final jumper wire to connect the ground leg of the LED to the negative (ground) power bus.

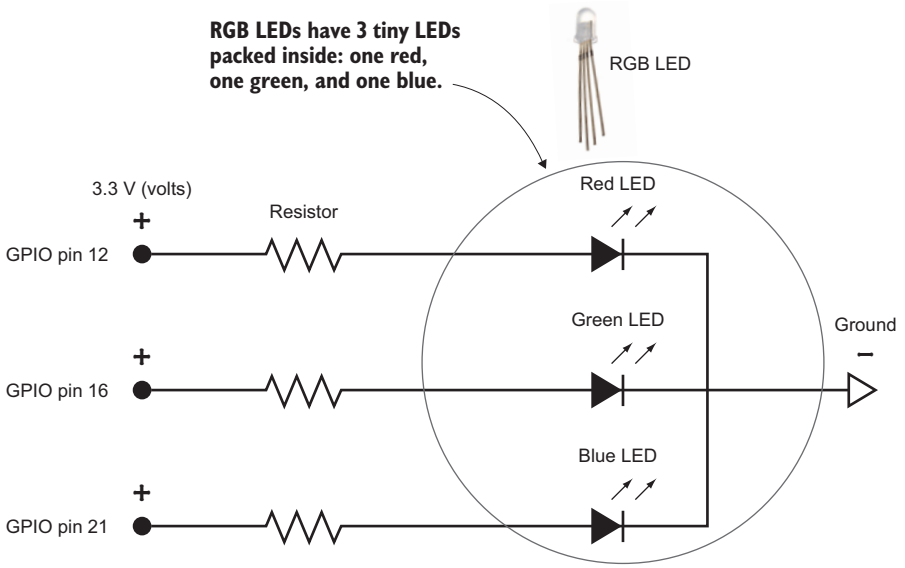


Figure 7.4 Circuit diagram for the Light Up Guessing Game project

When it's done, the circuit will look like what you see in figure 7.5. Let's walk through the steps to build this circuit.

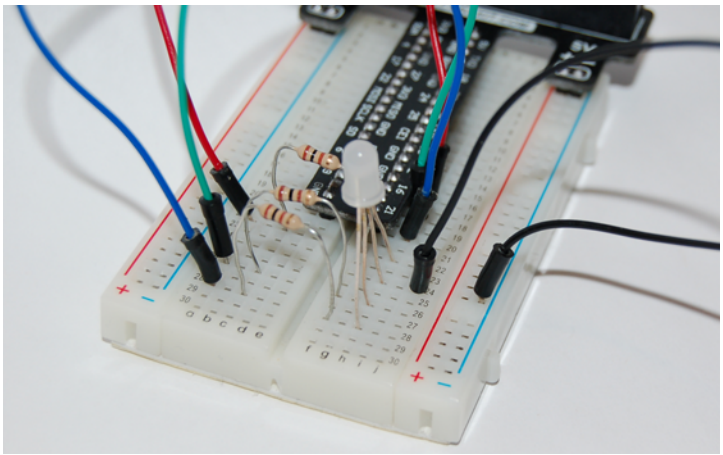


Figure 7.5 The RGB LED circuit you're building on the breadboard uses GPIO pins 12, 16, and 21 to power the LEDs. The light won't turn on until you program the voltage to come out of the pins.

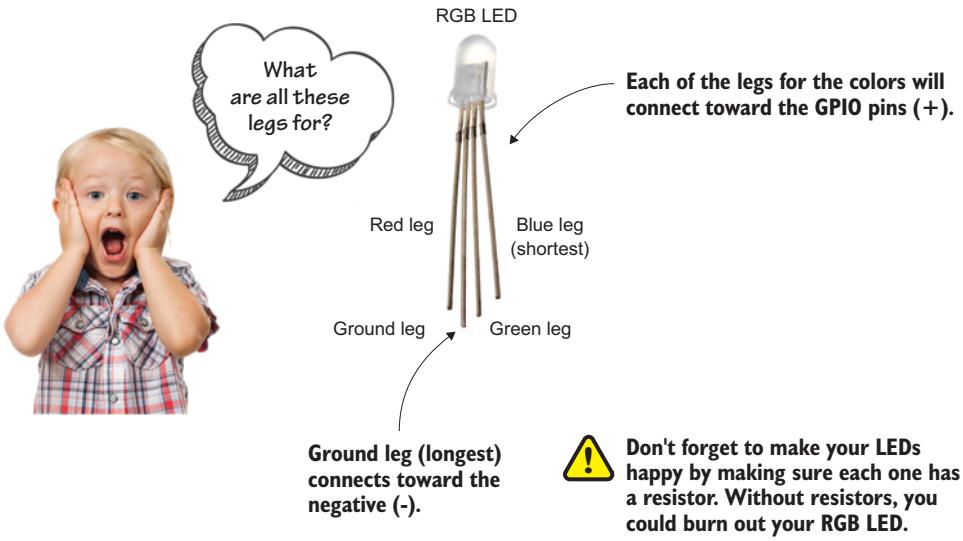


Figure 7.6 The RGB LED has lots of legs! The longest leg is the ground. The other ones are for red, green, and blue. This applies to what is called a *common cathode* RGB LED, which is what comes in Pi kits and what you'll find most commonly at electronics suppliers.

STEP 1. ADD THE RGB LED

Before you can add it to the breadboard, let's look a bit closer at the RGB LED. Remember that there are three tiny LEDs (red, green, and blue) inside it. You need to be able to figure out which leg is which color and which one is ground. Figure 7.6 is a handy reference.

> NOTE You'll need to bend the RGB LED's legs quite a bit to get them into the holes on the breadboard. Try to bend them to line up with the holes, and slowly push the legs in all at once.

Grab your RGB LED, and let's insert it into the breadboard. You're going to put it in rows 22, 24, 26, and 28 along column h on the breadboard. Here's where to connect the legs:

- Red leg into hole BB22h
- Ground leg (longest leg) into hole BB24h
- Green leg into hole BB26h
- Blue leg (shortest) into hole BB28h

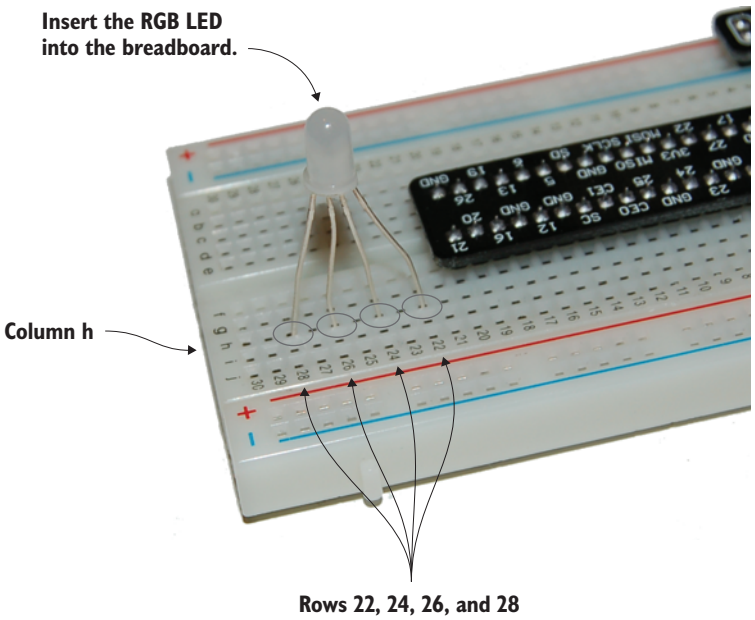


Figure 7.7 Bend the legs of the RGB LED, and insert it into the breadboard at BB22h, BB24h, BB26h, and BB28h. The longest leg goes into hole BB24h.

When it's inserted, it will look like Figure 7.7. Double-check that it's pushed down into the breadboard so all the legs will make a good connection.

Good job! You just completed the trickiest part.

STEP 2. CONNECT THE GPIO JUMPER WIRES

The breakout board has numbers on it that refer to the Raspberry Pi's GPIO numbering system. Remember that we refer to GPIO pins by adding *GPIO* before the number of the pin. So if we're talking about GPIO pin 12, it's GPIO12.

Question: What hole on your breadboard is next to *GPIO12* (GPIO pin 12)?

Answer: Look closely, and you'll see that the holes next to it are *BB16i* and *BB16j*.

> NOTE The color of the jumper wires doesn't matter, but it's sometimes helpful to pick ones that match the colors of the LED legs. When you're troubleshooting problems, that can help you easily remember which GPIO pin is controlling each color of light coming out of the RGB LED.

Now that you've located the holes near the GPIO pins, you can start connecting jumper wires as follows:

- Jumper wire from BB16j to BB22a (connects GP12 to the red leg of the RGB LED)
- Jumper wire from BB18j to BB26a (connects GP16 to the green leg of the RGB LED)
- Jumper wire from BB20j to BB28a (connects GP21 to the blue leg of the RGB LED)

When you've added the wires, the circuit will look like figure 7.8.

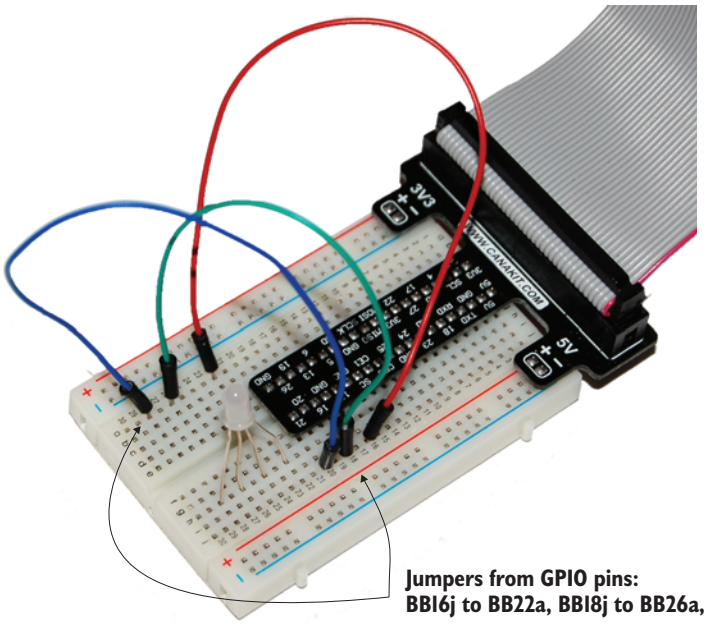


Figure 7.8 The jumpers connect the GPIO pins from your Pi to the RGB LED. If you have an earlier model Pi, you can use other GPIO pins. Just remember which ones you're using, and use these numbers when you program the Pi to turn the GPIO pins on and off.

STEP 3. ADD THE THREE RESISTORS

It's time to connect your 180 ohm resistors![1] They should have bands of brown, grey, and brown, followed by a fourth gold or silver band. Remember that electricity will flow either way through a resistor, so the way you connect it doesn't matter. Figure 7.9 is a handy diagram that reminds you how you can figure out the value of a resistor by using the colored bands.

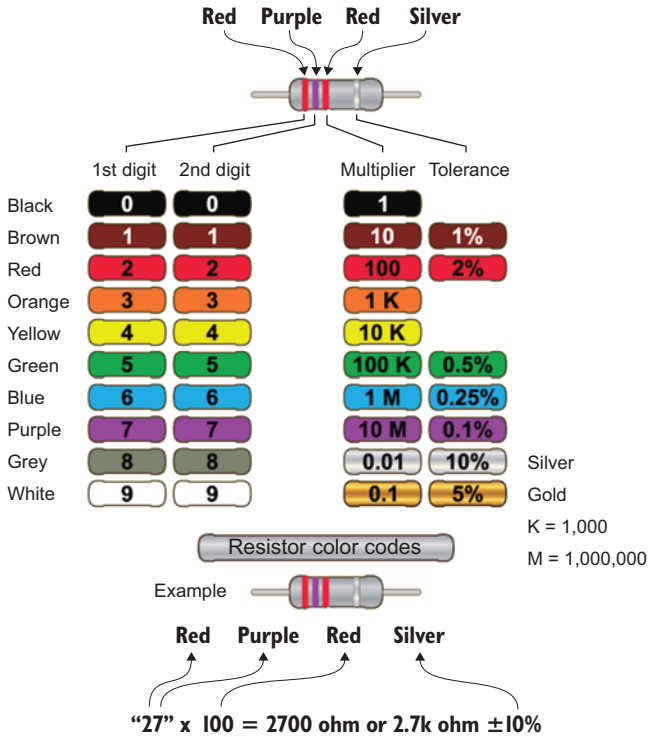


Figure 7.9 The colored bands on a resistor tell you how much resistance the resistor has. For this project, you want a brown (1), grey (8), brown (×10) resistor, or 18 × 10 = 180 ohm resistor. Don't have one? Any resistor between about 100 and 300 ohms should work well.

[1] This is a safe value that won't risk damage to your Pi and will keep things simple. For those of you who are into precision, technically you might want to use slightly different resistors for each color LED (red, green, and blue), because each one requires a different amount of electrical current (amps) to make it shine. Check out some of the online resistor calculators and Pi forums on RGB LEDs if you're interested.

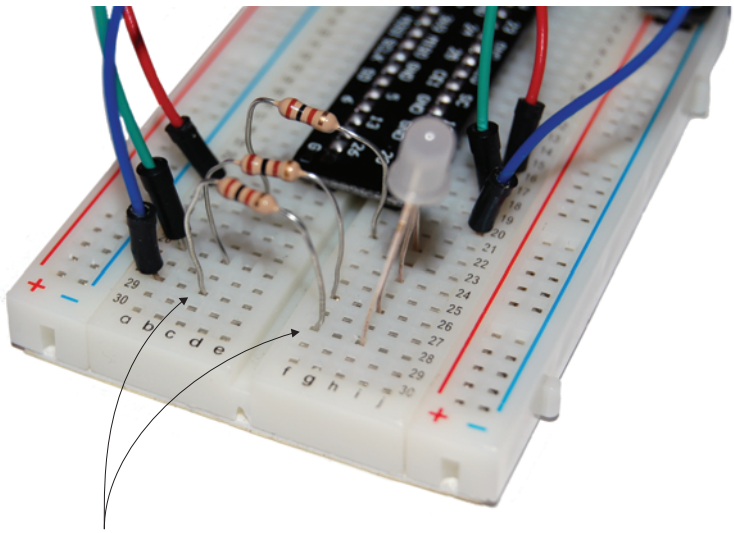


Figure 7.10 Add your resistors! Make sure you push them down into the breadboard holes. If you don't like them sticking up so high, you can trim the ends using wire cutters.

Connect the resistors as follows:

- Insert one end of the first resistor into BB22c and the other end into BB22f.
- Insert one end of the second resistor into BB26c and the other end into BB26f.
- Insert one end of the third resistor into BB28c and the other end into BB28f.

Once they're added, you'll have something that looks like figure 7.10. Now you're ready for the final step!

STEP 4. ADD THE JUMPER TO GROUND

Remember that a ground rail runs vertically along the right side of the breadboard, with a blue stripe next to it. Add a jumper from BB24j to

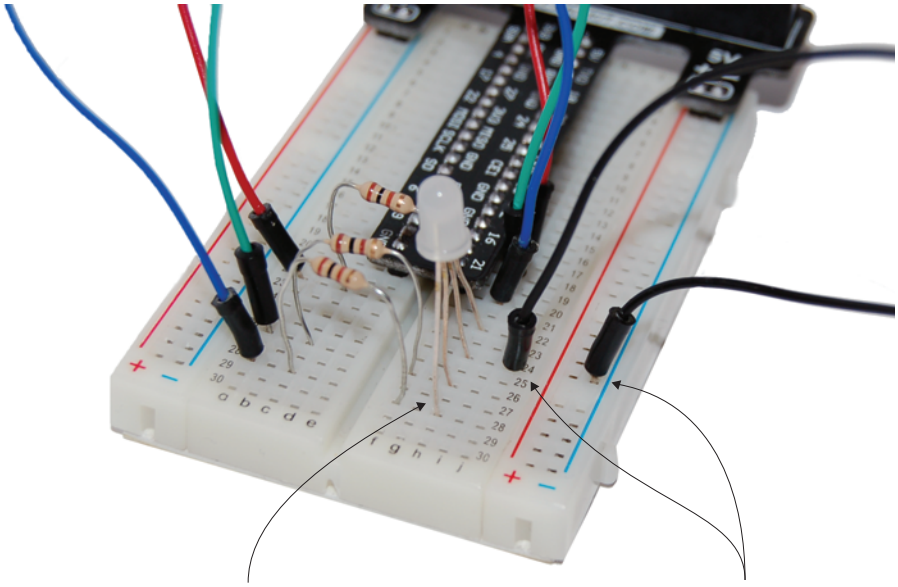


Figure 7.11 The jumper is added to connect the ground of the RGB LED to the ground of the Raspberry Pi. The jumper can connect anywhere along the ground rail (it usually has a blue stripe running next to it).

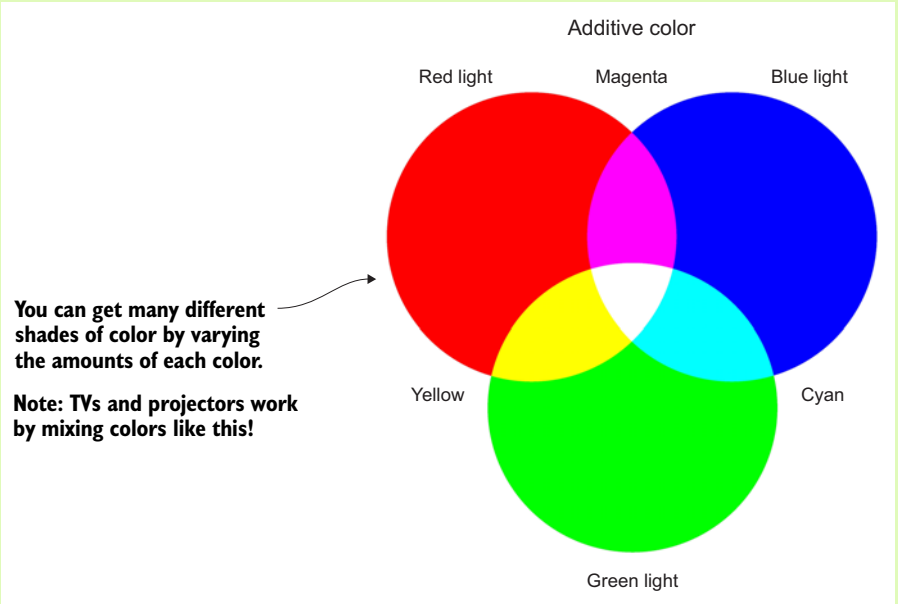
the negative (-) power bus or ground rail (any hole next to the blue stripe will do). Figure 7.11 shows how it looks.

Wahoo! You've completed the RGB circuit on the breadboard. With the circuit complete, it's time to write your program so you can test it.

Color mixing with an RGB LED

You can program your RGB LED to light up red, green, or blue by turning on or off GPIO pins 12, 16, and 21. But RGB LEDs can make more colors by mixing different amounts of red, green, and blue light. For example, you can combine equal amounts of red and blue light to make a nice magenta color. Or to make your LED yellow, you can combine equal amounts of green and red. Televisions work on the same principle. This concept, called *additive color*, means mixing varying amounts of different colors of light to make new colors.

(continued)



Wait! Your Pi can only turn LEDs on or off (you set them to `HIGH` or `LOW`)! How can you make something like a raspberry red color that might be 80% red and 20% blue? It's possible, but you'll need to learn how to very quickly pulse your Pi's GPIO output. This is called *pulse width modulation (PWM)*. Check online for information on how you can use the `RPi.GPIO` module to do PWM and create almost any shade of color you want.

Software: LEDGuessingGame program

You're creating a game to guess a magic number. As mentioned at the start of the chapter, you'll design the game play based on these simple rules (feel free to change them to your liking):

- The magic number is a randomly generated number between 1 and 20.
- The player is given five tries to guess the number correctly.
- If they guess correctly, the RGB LED flashes green.
- If they guess too high, the RGB LED flashes red.
- If they guess too low, the RGB LED flashes blue.

- After five guesses, the game is over.
- The player is given the choice to play again.

As you've seen in earlier chapters, programming is often about breaking down complex problems into smaller ones and then solving them. Let's start by laying out a quick diagram outlining what the program should do (see figure 7.12).

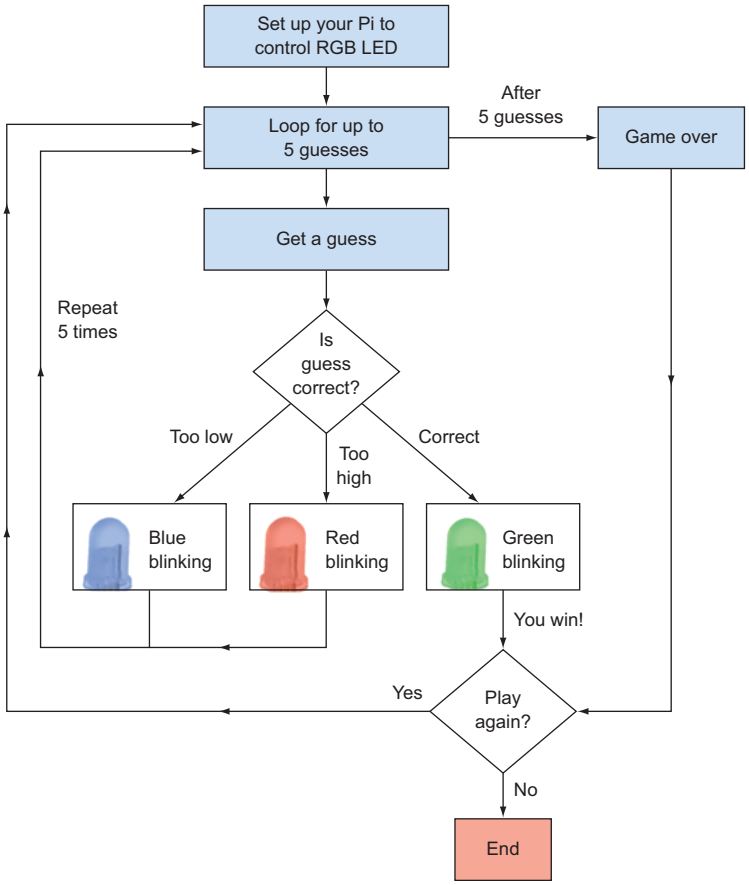


Figure 7.12 Flow diagram showing how the guessing game should work. Notice how you're blinking LEDs if the guess is too low, too high, or correct. You also give the player the choice of whether they'd like to play again.

As you approach this program, let's see if you can simplify the code by organizing it into *functions*, especially when you have chunks of code that can be easily separated. Remember that you can use functions to organize your code and simplify it. You'll create three functions to handle each of the flashing lights, to simplify the main part of your program:

- `flash_red` — Flashes the RGB LED red
- `flash_blue` — Flashes the RGB LED blue
- `flash_green` — Flashes the RGB LED green

You'll also create a function to display a message when the game is over.

Now that you have a plan, let's code it in this order:

1. Import libraries, create the flashing and game-over functions, and set up the GPIO pins for RGB LED output.
2. Display the title and introduction, create a loop, and get and check up to five guesses.
3. Add logic to allow the user to decide if they want to play again.

Let's begin! Open IDLE by choosing Python 3 under Menu > Programming. This opens IDLE to the Python 3.x Shell. In the Python Shell, start a new program by pressing Ctrl-N or selecting File > New Window.

Setting up the GPIO pins for the RGB LED

In the IDLE 3 text editor, you'll first load the Python libraries you need, create functions, and prepare your Pi to send electricity to the RGB LED (see figure 7.13).

SETTING UP YOUR PI'S GPIO PINS

You need to get your Pi ready for output to the GPIO pins and tell the Pi which pins you plan to use (see listing 7.1). If you recall from the earlier wiring, you're using these pins to control the three LEDs that are inside the RGB LED:

- GP12 for the red LED
- GP16 for the green LED
- GP21 for the blue LED

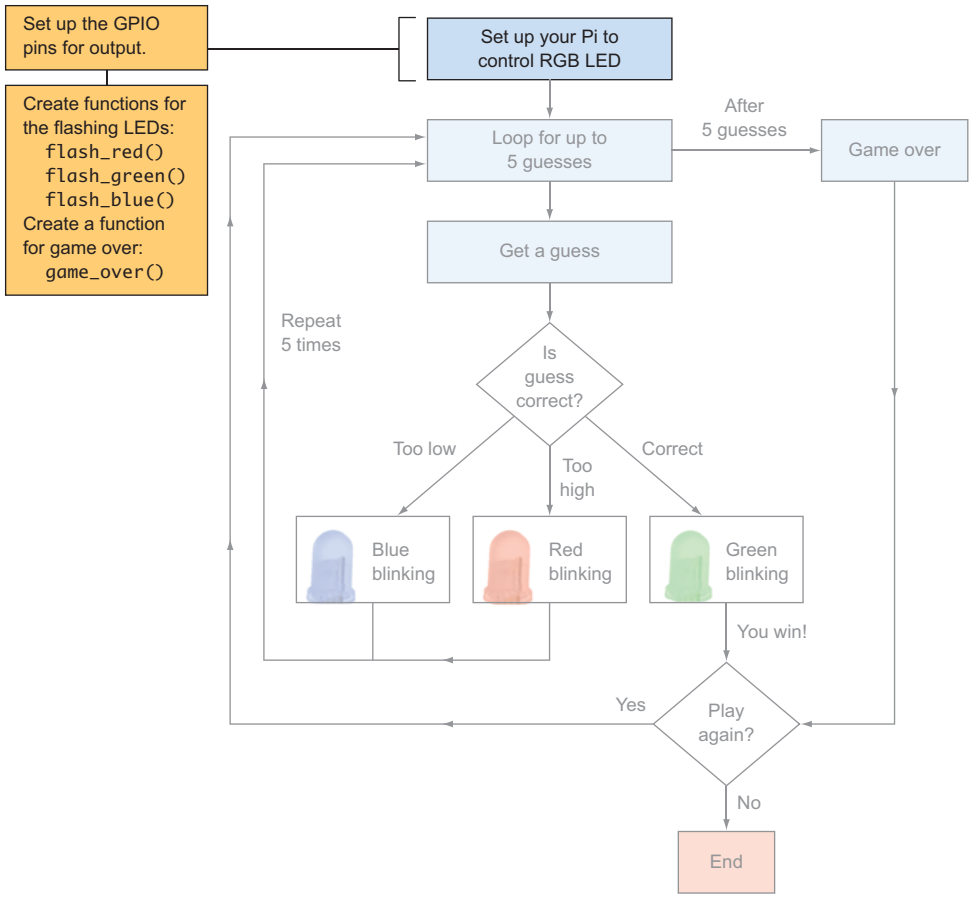


Figure 7.13 The program starts by importing the Python libraries you'll need to use, setting up your Pi's GPIO pins for lighting up the LEDs, and defining the functions you'll need.

Later, you'll write the code to control those pins. Let's start by importing the GPIO library for the Raspberry Pi and setting up the GPIO pins so they can output a voltage to control the RGB LED.

Listing 7.1 Setting up the Pi's GPIO pins

```
# Light Up Guessing Game
# Ryan Heitz
```

```
# importing the libraries we need
import RPi.GPIO as GPIO
import time
import random


#Tell the Pi we want to use a breakout board
GPIO.setmode(GPIO.BCM)

# Create variables for the pins used for LEDs
LED_pin_red = 12
LED_pin_green = 16
LED_pin_blue = 21

# Blink speed in seconds
blink_time = 0.25

# Tell the Pi which Pins we will use
# Set them up as OUT pins (send electricity out)
GPIO.setup(LED_pin_red,GPIO.OUT)
GPIO.setup(LED_pin_green,GPIO.OUT)
GPIO.setup(LED_pin_blue,GPIO.OUT)
```

Import several libraries you'll need later.

Pick which GPIO pins you'll use to light the LEDs.

Create a variable to store how long the light should blink on and off.

Tell your Pi to set up three GPIO pins for output.

Great! You've started by importing the `time` and `random` libraries, because you'll need them to flash the LED and help you generate a random number when the game starts. You define variables for the pins you're using and even add a variable, `BlinkTime`, that says how much time you'll blink the light on and off. Finally, you tell your Pi that you want to use three pins as output. Now let's write the functions.

CREATING FUNCTIONS TO SIMPLIFY THE CODE

You need three functions to flash the three LEDs inside the RGB LED and one for game over. Name the flashing functions `flash_red`, `flash_blue`, and `flash_green`, as shown in the following listing.

Listing 7.2 Functions that flash LEDs different colors

```
# Blinks an LED.
def flash_red():
    for i in range(1,6): #Blink on and off 5 times
        # Turning on LEDs
```

Use a for loop to flash the LED five times.

Define the name of the flashing function.

```
        GPIO.output(LED_pin_red, GPIO.HIGH)
        time.sleep(blink_time)
        GPIO.output(LED_pin_red, GPIO.LOW)
        time.sleep(blink_time)

def flash_green():
    for i in range(1,6): #Blink on and off 5 times
        # Turning on LEDs
        GPIO.output(LED_pin_green, GPIO.HIGH)
        time.sleep(blink_time)
        GPIO.output(LED_pin_green, GPIO.LOW)
        time.sleep(blink_time)

def flash_blue():
    for i in range(1,6): #Blink on and off 5 times
        # Turning on LEDs
        GPIO.output(LED_pin_blue, GPIO.HIGH)
        time.sleep(blink_time)
        GPIO.output(LED_pin_blue, GPIO.LOW)
        time.sleep(blink_time)

# An ending to the game if they don't guess it
def game_over():
    print("You lost!")
    print("Better luck next time!")
    time.sleep(2)
```

Tell the Pi to start (GPIO.HIGH) outputting voltage to the GPIO pin.

Pause the program to blink the light.

Tell the Pi to stop (GPIO.LOW) outputting voltage to the GPIO pin.

Display messages, and pause for 2 seconds.

In the code, you create four functions:

- `flash_red()`
- `flash_green()`
- `flash_blue()`
- `game_over()`

The three flashing functions blink a different color LED in the RGB LED. The blinking is created by using a `for` loop and the `sleep` function while you switch the output from the GPIO pin from `HIGH` (on) to `LOW` (off). Think of this as being like standing at a light switch and flipping it on and then off, five times.

Before you go any farther, save the program as LEDGuessingGame.py in your home folder.

When to use functions

Believe it or not, we don't always know when to create a function. The ability to figure that out is a skill that comes with experience in writing programs and seeing patterns. Here are some tips for deciding what to make a function:

- Is there a group of instructions that you'll need to use over and over again, with little variation?
- Do you have large blocks of code that make your programs hard to read?

Functions can simplify your code and make it easier to update.

REFACTORING YOUR FUNCTIONS

Did you notice that the functions for flashing the LEDs are very similar? Most of the code in each function is the same except for the GPIO pin, so let's see if you can improve this code to make it simpler. This process of simplifying code is called *refactoring*.

What if you rewrote the three functions as a single function, as shown in listing 7.3? This new function takes one parameter, `LED_pin`, that represents the number of the GPIO pin you want to control. It can be any one of the GPIO pins you're using for the colors of the RGB LED. For example, if `LED_pin` is 16, this corresponds to GPIO pin 16, which should blink the green light.

Listing 7.3 Refactoring the three flashing functions to a single function

```
# Blinks an LED.
def flash(LED_pin):
    for i in range(1,6): #Blink on and off 5 times
        # Turning on LEDs
        GPIO.output(LED_pin, GPIO.HIGH)
        time.sleep(blink_time)
        GPIO.output(LED_pin, GPIO.LOW)
        time.sleep(blink_time)
```

The function takes one parameter as input (the GPIO pin number).

Turn the signal to the LED on and off.

In this case, you're refactoring a set of functions that are very similar to a single function that takes a parameter (`LED_pin`). This parameter makes the function more flexible or dynamic so it can take the place of the three separate functions.

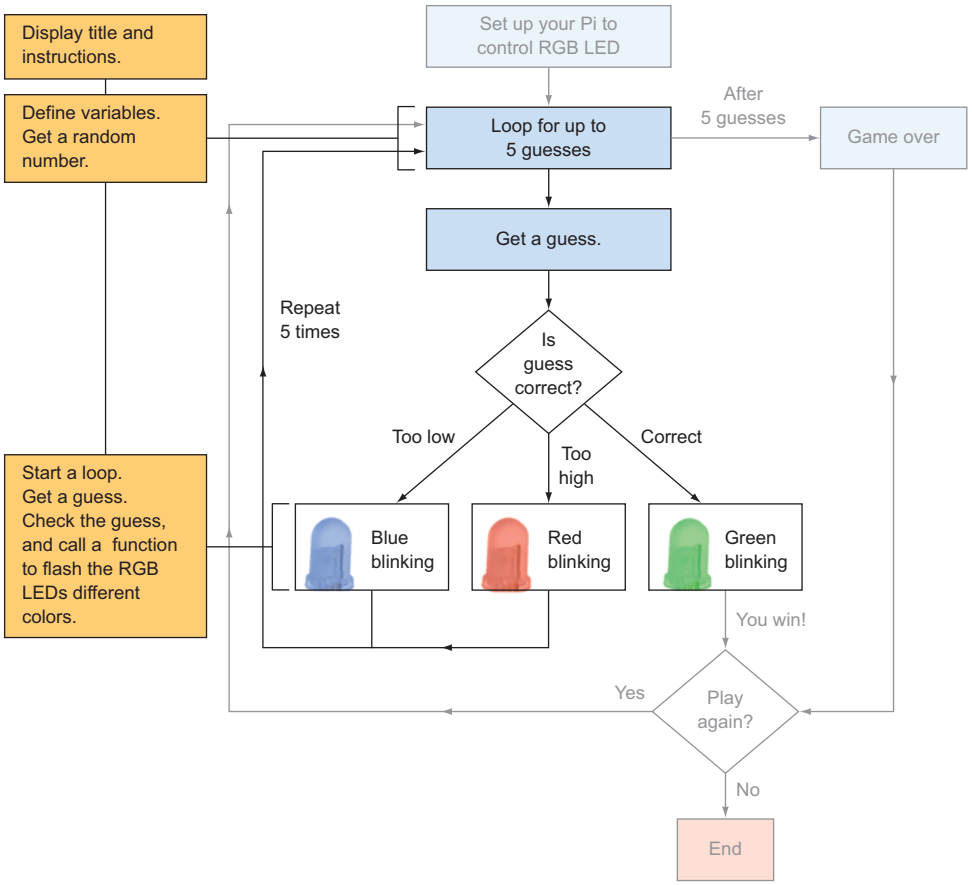


Figure 7.14 After displaying the game title and instructions, you need to define variables to store important game information, including a random number the player is trying to guess. The main loop in the game is repeated to allow the user to make five guesses; it also blinks the lights.

Main game loop and logic

The next part of the program creates the main game loop (see figure 7.14). You'll do the following:

- Set up the game.
- Display the title and instructions for the person playing.
- Create some variables, and get a random number.
- Create the loop and guessing logic.

GAME SETUP

Let's look at some of the variables you'll need for the game:

- `number_in_my_head` holds a random number (an integer between 1 and 20) that the player is trying to guess.
- `count_guesses` helps you count and keep track of how many guesses the player has made.
- `play_again` tracks the status of whether the player wants to play again. You'll use a Boolean type for this, because it should always be True (yes, let's play again) or False (no, let's not play again).

The next listing adds these three variables and sets them up. You also create and display the title and game instructions.

Listing 7.4 Creating variables and displaying the game title and instructions

```
# A random number for our game
number_in_my_head = random.randint(1,20)
count_guesses = 1  # Counter for the number of guesses

# Used to keep track of whether they want to play again
play_again = True

title = """
**************************************************************************
                          Light Up Guessing Game
**************************************************************************
"""
print(title)

intro = """
Game Play:
I'm thinking of a number between 1 and 20. You have five guesses to
  guess it.
After each guess, my light will blink.

  Red ---> Your guess is too high!
  Green ---> Your guess is correct!
  Blue --> Your guess is too low
"""
```

Fantastic! The variables set the stage for the guessing-game logic. It's a lot like the foundation of a house—you need it in order to build the rest.

Guessing Game Loop and logic

The code features two loops, one inside the other. The outer loop gives the user the option of playing again—we'll call this the Play Again Loop. Within that loop is another that gives the player five guesses—we'll call this the Guessing Game Loop.

The main game loop involves getting a guess, checking the guess, blinking the RGB LED the appropriate color, and then repeating until the player guesses right or has used all five guesses. The next listing shows the program for the Guessing Game Loop and the logic for checking guesses.

Listing 7.5 Guessing Game Loop

Start the game loop that gives the player 5 chances to guess correctly.

Display a prompt for a user to enter their guess.

Call the flash function and tell it to flash the RGB LED a certain color.

Exit (break out of) the game loop if the player guesses correctly.

Call after the player has guessed 5 times.

```
while count_guesses < 6:
    guess = input("Guess " + str(count_guesses) + ": ")
    guess = int(guess)  # Convert the input string to an integer
    count_guesses += 1  # Add one to the number of guesses
    ➥ to keep track
    if guess == number_in_my_head: # Guessed it correctly
        flash(LED_pin_green)
        print("You won!")
        break # Breaks out of loop
    elif guess > number_in_my_head: # Guess too high
        flash(LED_pin_red)
    elif guess < number_in_my_head: # Guess too low
        flash(LED_pin_blue)
else: # For the while loop, it happens when the while condition
➥ isn't True
    game_over()
# End of game
```

The Guessing Game Loop contains the logic to

- Keep track of the number of guesses.
- Get a guess.
- Check to see if a guess is correct, too high, or too low.

Where is the logic for responding to the player? It's in the loop. Each time you get a guess, a series of `if`/`elif` statements checks whether the guess is correct, too high, or too low. Based on which of those cases is True, the `flash()` function is called to flash the appropriately colored LED on and off. If the user guesses the number correctly, the RGB LED will flash green, and then the `break` command will exit the `while` loop.

Notice that you add an `else` statement to the `while` loop. When the number of guesses has been exceeded (`count_guesses` is greater than 5), the `else` statement is triggered and the `game_over` function is called. The `else` block only happens when the `while` condition is checked and is False (in this case, when the number of guesses has exceeded 5).

In the next section, you'll see how to give the player the option of playing again.

Adding the Play Again Loop and logic

You want to add a feature to the game that lets the user choose whether they want to play again. To do this, you need another loop that goes around the Guessing Game Loop (see figure 7.15). The Play Again Loop needs to repeat the Guessing Game Loop as long as the user answers that they want to play again.

Listing 7.6 Play Again Loop

```
while play_again:
    print(intro)

    # Guessing Game Loop
    while count_guesses < 6:
        # Loop code hidden
    else
        game_over()
    # End of Guessing Game Loop
    answer = input("Would you like to play again [Y/N]? ")
    if answer.upper() ==  "Y":
        # Starting over. Get a new random number and reset the counter

        number_in_my_head = random.randint(1,20)
        count_guesses = 1
```

Start the Play Again Loop that repeats as long as play_again is True.

Print instructions.

Ask the user if they want to play again.

Get a new random number.

Reset the number of guesses to 1.

```
    else:
        play_again = False
print("Good bye!")
GPIO.cleanup()
```

Set play_again to False, which causes the Play Again Loop to end.

Reset the GPIO pins used in this program (set them back to input).

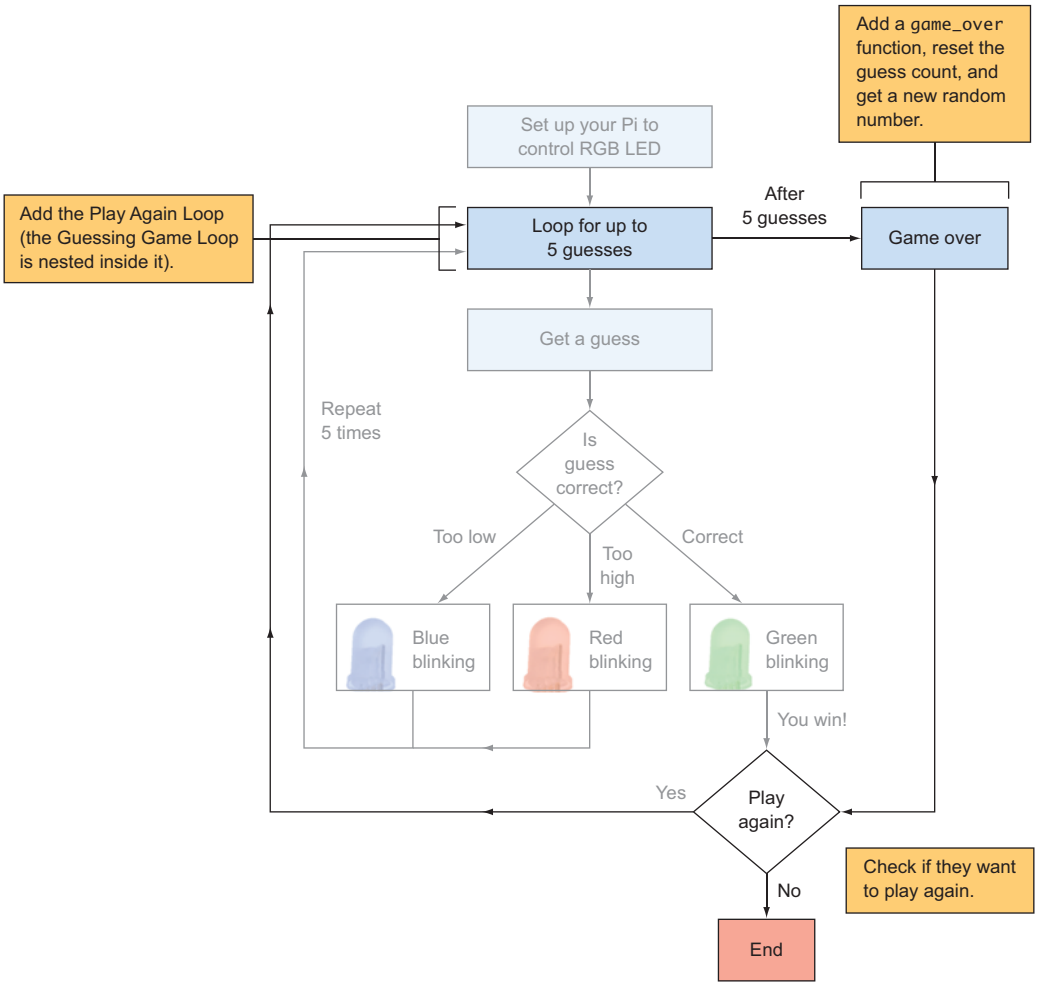


Figure 7.15 The Play Again Loop is wrapped around the Guessing Game Loop. After the player has exhausted their guesses or guessed the number correctly, they're asked if they want to play again. Depending on their answer, the game will either start over or end.

Awesome job! You have put together a circuit to control an RGB LED and written the Python code to make a game interact with it. Now, let's test it.

Playing the game

Save the code as LEDGuessingGame.py, and try running it. Select Run > Run Module (or press F5) from the IDLE text editor to run your program. If you have an older version of Raspbian (prior to October 2015), open Terminal and enter the following command:

```
pi@raspberrypi ~ $ sudo python3 LEDGuessingGame.py
```

Excellent! You should see your guessing game start up. Let's test it to see if it works. Try seeing if you can guess the number. Try getting it wrong, just to make sure the `game_over` function works.

> **NOTE** Remember that any programs that use GPIO pins must be run from the Raspbian command prompt as the superuser (or root). The `sudo` command lets you do this. If you try running the program at the Python Shell in IDLE, then you'll get the error that ends "`RuntimeError: No access to /dev/mem. Try running as root!`"

Troubleshooting

If the lights aren't blinking after each guess is made, here are some things you can check:

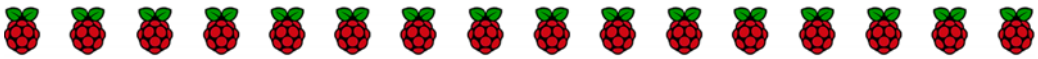
- Check the circuit on the breadboard. Is the ribbon cable connected properly, with the first wire connected toward the edge of the Pi, away from the USB ports?
- Double-check that the jumper, RGB LED, and resistors are connected to the correct holes on the breadboard. Could your RGB LED be inserted the wrong way (the shorter legs go toward the negative or ground side)? Try turning it around if you aren't sure.
- Look through your Python program for errors. If necessary, edit the program to add some `print` statements so you can see which parts are working. For example, in the inner loop that handles the five guesses, you can use the `print` function to display the value of `count_guesses`:

```
print(count_guesses)
```

- Try adding a `print` message in the `flash` function so you're sure it's being called. For example, you could add

```
print("Blinking the LED")
```

If you've enjoyed playing your game, try some additional challenges to increase the fun factor!



Challenges

These challenges use the RGB LED that you've already wired up. If you can't figure them out, check appendix C for hints and solutions.

Game winner

Write a function in the game that creates a flashing animation whenever the user correctly guesses the number. For example, you could try quickly flashing the RGB LED different colors.

Easter egg

Was the last one too easy? Well, try this: create an Easter egg in your game. Create logic so that if someone types in a certain word (maybe *Spam*), the program displays a secret message and flashes the light in a crazy way.

Warmer and colder

Expand the logic of your program to make the speed of the blinking indicate whether the player's guess is close to or far away from the correct answer. As a hint, think about the blinking speed you've set. Let's say a guess is off by 10 (the player guesses 15, and the magic number is 5). You want the light to blink slowly. You can take the difference (ignore any negative signs) and divide it by 10. This will make the blinking speed one-tenth of the difference, or once every second if you're off by 10 (pretty slow). If the player's guess is off by 2, the light

will blink every two-tenths of a second (pretty fast). This way, the blinking speed tells the player if their guess is close or far away.

Darth Vader surprise

Let's see if you can get an image of Darth Vader to pop up if the player doesn't correctly guess the number. Here's a hint to get you started. Install the Linux image-viewing software called fim,[2] a program that allows you to open images from the Raspbian command line. To install fim, make sure your Pi is connected to the internet, and then open Terminal and use the following command:

```
pi@raspberrypi ~ $ sudo apt-get -y install fim
```

Next, download an image of Darth Vader and have the game display it on the screen. Let's say you've downloaded an image called Darth_Vader.jpg. You can display it with these commands in Python:

```
import os
os.system("fim Darth_Vader.jpg")
```

Good luck! May the Force be with you!

Summary

In this chapter, you learned that

- Pis can respond in rich and exciting ways by interacting through the GPIO pins in your programs.
- Functions, loops, and conditional statements can be combined with your Pi's output capabilities to create programs that react to people and the environment.
- RGB LEDs are very cool because they can make different colors and are actually three LEDs packed into one small package.
- A `while` loop can have an `else` statement that allows you to control what happens when the loop condition is no longer true.

[2] fim is the improved version of fbi, image-viewing software for Linux that can be run from the command line.

- A play again loop can be wrapped around a main game loop to allow users to play the game over and over again.
- *Refactoring* is a fancy word that just means simplifying or shortening your code by looking for ways to make it more efficient. Be careful, though—you don't want to simplify something so much that it becomes too hard to understand (remember the Zen of Python)!

8

DJ Raspi

In this chapter, you'll be

- *Giving your Pi the ability to respond to input signals by making it interact with you in response to button presses*
- *Learning about electronic buttons and how to build circuits on a breadboard with them*
- *Running Raspbian operating system commands so your programs can play music, show videos, and more*
- *Using Python to store sets of information called* ***lists***
- *Exploring how you can play sounds on your Pi and make your Pi into a music machine*

We don't think about our five senses (taste, smell, touch, hearing, and sight), but without them we wouldn't be able to feel, know, and interact with the world around us. Think of your Pi as a person who, until now, has had a limited set of senses. So far, your Pi has only been able to respond to keyboard keys being pressed and mouse clicks.

Like a mad scientist bringing something to life, in this chapter you're going to embark on a project to wire up a new sense of touch for your Pi.

Okay, maybe it won't be as crazy as creating a bionic creature, but a button gives your Pi a sense of touch. You'll wire a couple buttons to the Pi's GPIO pins (recall that GPIO stands for general-purpose input/output, so this is how your Pi can sense and affect the environment). Then you'll program your Pi to react to button presses. Exciting times are ahead!

This project is a small glimpse of all the different senses you could possibly give your Pi. Electronic components that can detect the environment around them are called *sensors*. A button is one of the simplest sensors, because it can detect touch. What other sensors could you add? How about some of these ideas:

- A camera that can track a ball or face using special software called *computer vision* that can recognize objects (this is similar to how a Microsoft Kinect works)
- Super-human capabilities like a proximity sensor to detect when someone is walking nearby (like the ones used to trigger the doors to open at the grocery store)
- A microphone so it can hear

All this is possible with a Pi, some determination to figure it out, and a bit of fearlessness about trying new things.

Project overview

In this chapter, you'll turn your Pi into DJ Raspi—a musical computer that plays different sounds when you press buttons. You'll wire up two mini pushbuttons on your breadboard and figure out how to write the code to make the buttons play sounds. Later, if you want, you can add other sensors to your Pi and program them. This project will give you an example of how to work with input from sensors. Figure 8.1 shows the parts you'll need.

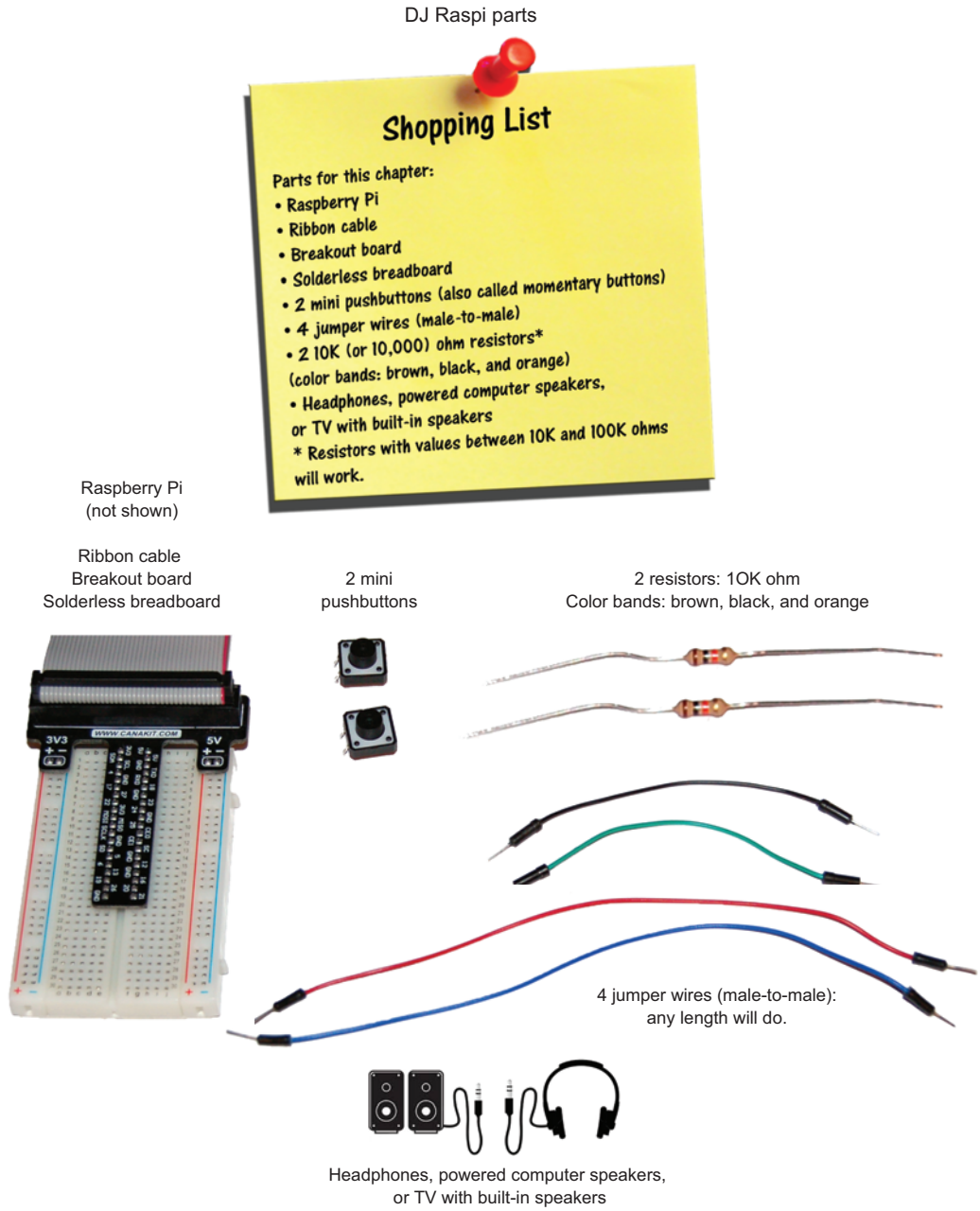


Figure 8.1 The DJ Raspi project requires several different parts to turn your Pi into a music player. The length and color of the jumper wires don't matter.

Gather the parts and get ready for some fun. You'll notice that some of them are the same as in chapters 6 and 7, but you'll also need a few new items. Most of these are included in Raspberry Pi starter kits, but you can find them at online electronics retailers as well. You'll approach this project in two parts: building the circuit (the hardware) and writing the program (the software). Let's go!

Setting up your Pi to play sounds

To start, let's get your Pi ready to play sounds. A Pi can output sounds through the headphone jack (also called the *3.5 mm audio port*) or through HDMI. Before you start, plug in your headphones, powered computer speakers, or, alternatively, a TV with built-in speakers connected via an HDMI cable.

All sounds aren't the same: audio formats

If you wanted to leave a secret message for someone, you could choose several different ways to make the message into a secret code. You could use different symbols to represent words, or you might substitute letters or shift letters around. There are many different ways to encode something.

Similarly, people have come up with many different ways to store sounds (or audio files). These ways (called *formats*) are different ways of compressing or encoding the information in a sound to make it easy to store on a computer or music player. Sometimes sounds are encoded so they will only work on certain music players.

Here are some common formats:

- *MP3*—The most common audio file format used in most audio players. The files end in .mp3.
- *WAV or WAVE*—Stands for Waveform Audio File Format. It's used on many Windows computers. These files end in .wav.
- *Ogg*—An open format that was developed for streaming applications. The files end in .ogg.

Each format uses a different method to compress or shrink a sound and make it smaller to store. The Pi has many different software applications for playing audio. Each one can play different formats. Check the Raspberry Pi forums if you want to learn more about the different players and what they're best for.

You'll be focusing on playing MP3s from your Pi, because that is a common audio file format. What can you use to play them?

OMXPlayer and MP3s

When you watch movies or listen to music on a computer, you may use iTunes or Windows Media Player. Raspbian has its own equivalent called OMXPlayer that can play sounds or videos. Lucky for you, it's capable of playing MP3 files (or MP3s)—one of the most common audio formats.

> DEFINITION OMXPlayer is a video and audio player that was created for Raspberry Pi.

If you don't have an MP3, you can test OMXPlayer using one of the sounds already on your Pi. There are quite a few MP3s in the folders included with the Scratch software. Open File Manager, and go to this folder to see some of them: /usr/share/scratch/Media/Sounds/Vocals/. In the folder, you'll see both MP3 and WAV format files (see figure 8.2).

Figure 8.2 When you install Raspbian on your Pi, it comes with Scratch, which has a number of sound files including vocals, sound effects, animal sounds, and drum beats.

To play an MP3 using OMXPlayer, open Terminal, and enter

```
pi@raspberrypi ~ $ omxplayer /usr/share/scratch/Media/Sounds/Vocals/
➥ Oooo-badada.mp3
```

You should hear a short music clip of a woman singing. Enjoy the song!

> NOTE In Terminal, pressing the up and down arrows cycles through previous commands. Press the up arrow once and then press Enter to run the last command again.

Fantastic! Your Pi can speak to you now.

Troubleshooting

What if you have speakers or headphones plugged in but don't hear anything? OMXPlayer should automatically detect whether to output the sound to the 3.5 mm audio output or HDMI. If it doesn't, try this command for the headphone jack (3.5 mm audio output):

```
pi@raspberrypi ~ $ omxplayer -o local /usr/share/scratch/Media/Sounds/
➥ Vocals/Oooo-badada.mp3
```

`-o` is a special switch or *flag* that lets OMXPlayer know that you want to tell it something. In this case, `-o` stands for *output*, and it tells OMXPlayer where you want to output the sound. In this case, you set it to `-o local`, which outputs sound to the 3.5 mm (headphone jack) output.

Switches (flags)

Switches, such as `-o` for output, act like options or special controls for a program. They're common when using the command-line interface. You can usually get a list of what switches a program has by making the command print out its help information. Most programs that you can run at the command line will give you a list of all switches or flags when you type the name of the program and then `-h`. The `-h` switch- stands for *help*. Try it with OMXPlayer:

```
pi@raspberrypi ~ $ omxplayer -h
```

You'll see a long list of options you can use to control how video and audio files are played. Try `-h` with other command-line programs to see what results you get.

If you need to specify sending the sound to speakers in your monitor, then use the `-o` switch and specify `hdmi` for output to the HDMI port:

```
pi@raspberrypi ~ $ omxplayer -o hdmi /usr/share/scratch/Media/Sounds/
➥ Vocals /Oooo-badada.mp3
```

Now that you know you can play music, let's build the circuit and write some code to create your DJ Raspi!

Hardware: building the circuit

Building time! You're building a circuit on your breadboard to detect or listen to buttons. When a button is pressed, your circuit will send electricity flowing to a GPIO pin on your Pi. You'll start by connecting the Pi's GPIO pins to the breadboard using the ribbon cable and GPIO breakout board. Refer back to chapter 6 (section 6.1) if you need to recall how to set this up.

> **A reminder about numbers**
> Like finding a seat in a stadium, we'll refer to the holes on a breadboard using the prefix *BB*. So the hole located in row 25, column a, is *BB25a*. Similarly, we'll refer to the Pi's GPIO pins using the prefix *GP* and then the pin number. So GPIO pin 24 is called *GP24* for short.

Wiring a button

Let's get busy wiring the buttons. There are many different types of buttons, but you'll be using a mini pushbutton (see figure 8.3). These buttons commonly come in Raspberry Pi kits along with jumper wires, resistors, and LEDs. If you need to purchase them, you can find them at many online electronics retailers in packs of 10 or 20 for less than the cost of a cheeseburger. With the parts gathered, let's assemble the circuit.

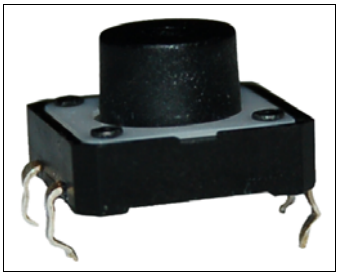


Figure 8.3 The mini pushbutton makes a nice clicking sound when you press the black button in the middle. Pressing it acts like closing a switch to complete a circuit.

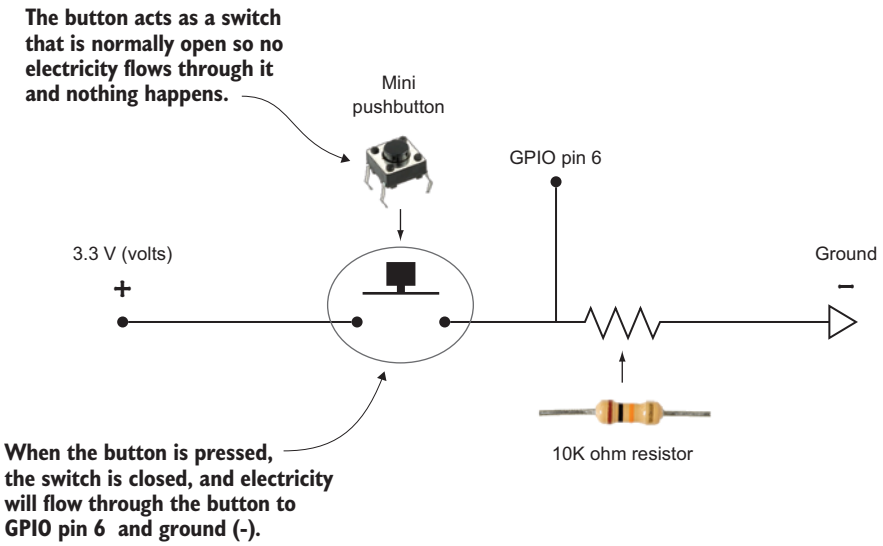


Figure 8.4 The circuit diagram for the first button in the DJ Raspi project shows how electricity will flow through the circuit. The button is a switch that allows electricity to flow to GP06 and ground (-) when it's pressed or closed.

Circuit sketch

The circuit diagram for the DJ Raspi is shown in figure 8.4. To listen to whether a button is being pressed, you'll have electricity (+3.3 V) flow from your Pi to the button. When the button is pressed, the electricity will flow through the button and then split. A small amount of electricity will flow to GPIO pin 6 (GP06) and the rest will flow through the 10K ohm resistor and then to ground (0 V). Let's put it together on the breadboard.

Let's build the button circuit on the breadboard and program your Pi to know when the button is being pressed. You'll give your Pi the ability to feel the button being pressed, by wiring up the button in this order:

1. Add the mini pushbutton to the breadboard.
2. Connect a jumper wire from 3.3 volts to the button. You'll use the positive power bus (+) that runs along the side of the breadboard.

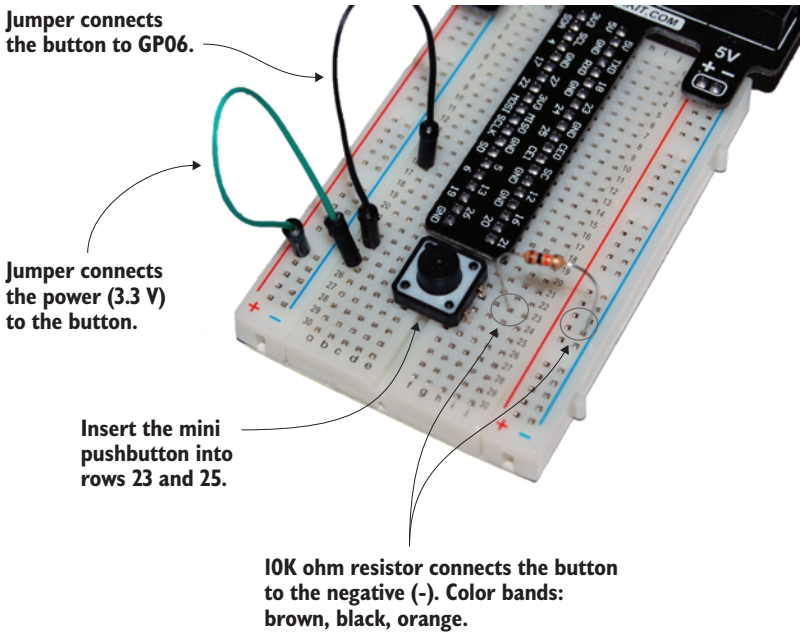


Figure 8.5 The mini pushbutton will have 3.3 volts connected to it from the positive power rail. When the button is pressed, power flows through the button and splits. Some electrical current goes to GP06 (GPIO pin 6), and the rest goes through the resistor and then to the negative power bus (-).

3. Add the resistor from the button to the negative power bus (-), also called ground.
4. Connect the second jumper wire from the button to GP06 (GPIO pin 6).

The completed circuit for one button will look like what you see in figure 8.5.

Don't forget, nothing will happen when you press the button. You have to program your Pi to react to this new-found sense of touch. Let's go through the steps to build the circuit:

STEP 1. ADD THE MINI PUSHBUTTON.

Let's look at how pushbuttons work before we go on. If you had X-ray goggles, you would see that the left and right legs at the top of the

button are connected. Similarly, the left and right legs along the bottom of the button are connected. The top and the bottom of the button aren't connected.

But when you press the button, figure 8.6 shows what happens. Pressing the button pushes down a small metal bar so that the top and bottom are connected. We say the switch is *closed*. When you let go of the button, the spring in the button pushes the metal bar back up, and the

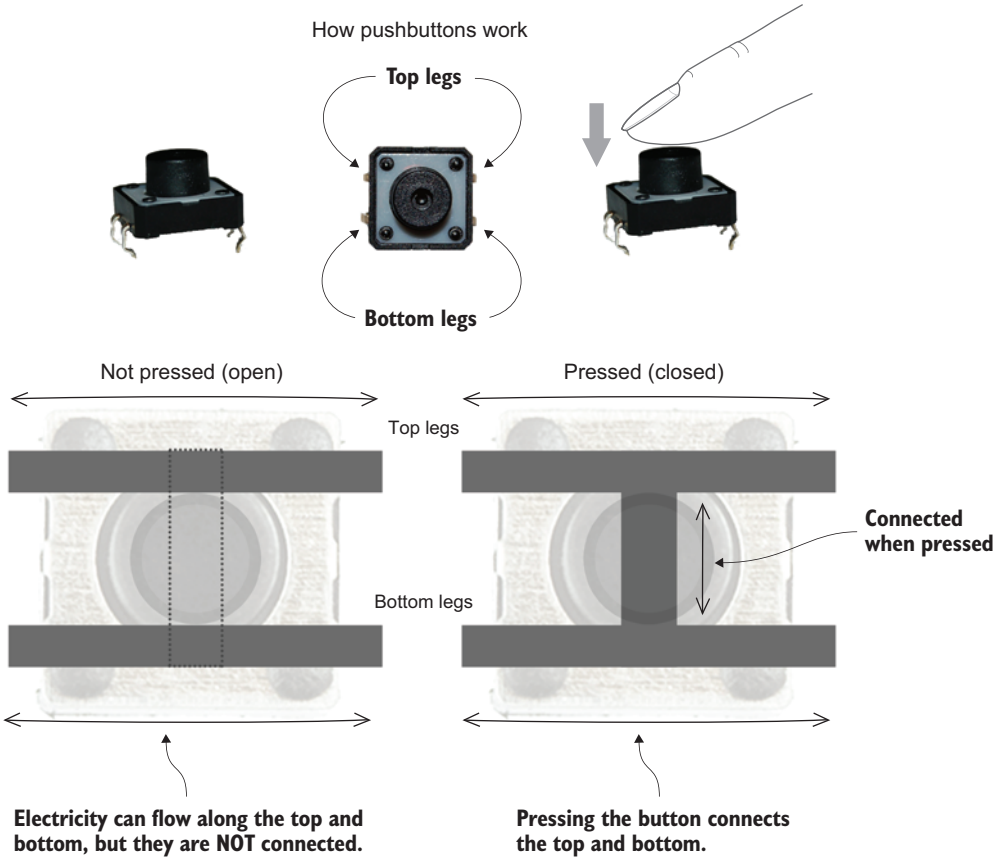


Figure 8.6 In a button, the legs are connected along the top and are separately connected along the bottom. When the button is pressed, the top and bottom are connected by a small metal bar.

switch is open again. Grab your mini pushbutton, and let's insert it into the breadboard.

> NOTE You'll need to push the button into the breadboard very firmly. If the button legs aren't lined up with the breadboard holes, you may accidentally bend some of the button legs. Don't worry—you can bend them back and try again. If a leg breaks off, use a new button.

You're going to put the button in rows 23 and 25 along columns d and g on the breadboard. Connect the legs:

- Top legs: *BB23d* and *BB23g*
- Bottom legs: *BB25d* and *BB25g*

When the button is inserted, it will look like figure 8.7. Double-check that it's pushed down into the breadboard so that all the legs will make a good connection. Good job—you just completed the trickiest part!

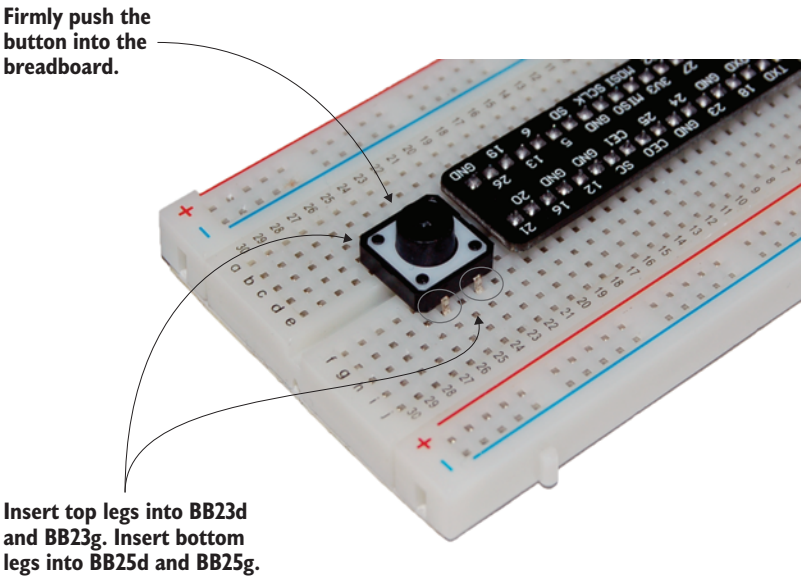


Figure 8.7 Align the pushbutton with the breadboard holes, and then press it down into the breadboard. Make sure you press it so the button legs are down into the breadboard holes and make a good connection. If you accidentally bend the legs, don't worry! Just bend them back and try again.

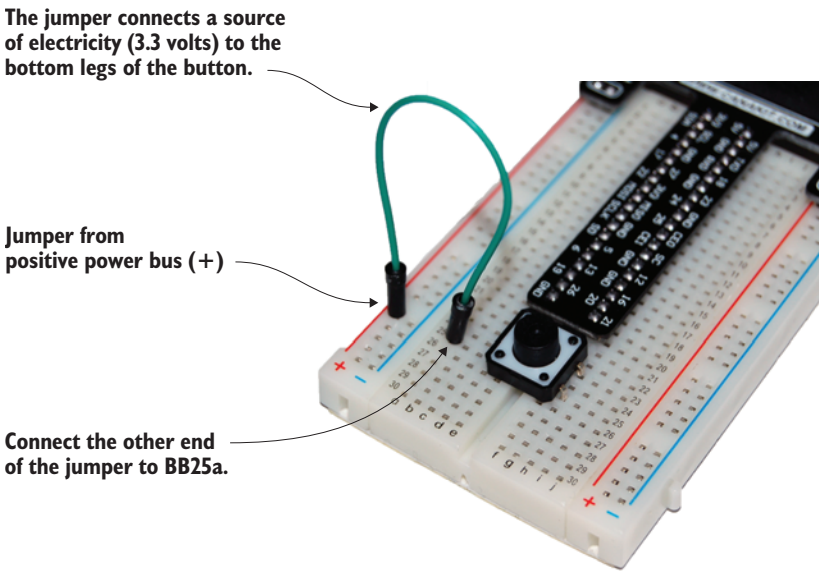


Figure 8.8 The jumper connects power (3.3 volts) to the bottom of the button.

STEP 2. CONNECT A JUMPER WIRE FROM 3.3 VOLTS TO THE BUTTON.

You need to connect the button to a source of electrical current. You'll use the positive power rail along the edge of the breadboard as the source of power (you could also directly connect the jumper to the 3V3 pin on the breakout board).

Connect the jumper wire from the *positive power bus* (+) to *BB25a*. Remember, you can connect the jumper to any hole along the power rail (it has a red line next to it). When you've added the wire, it will look like figure 8.8.

Fantastic! Now you have electricity reaching the bottom legs of the button.

STEP 3. ADD THE 10K OHM RESISTOR.

Time to connect your 10K ohm resistor. It has bands of brown, black, and orange followed by a fourth gold or silver band. *Remember that*

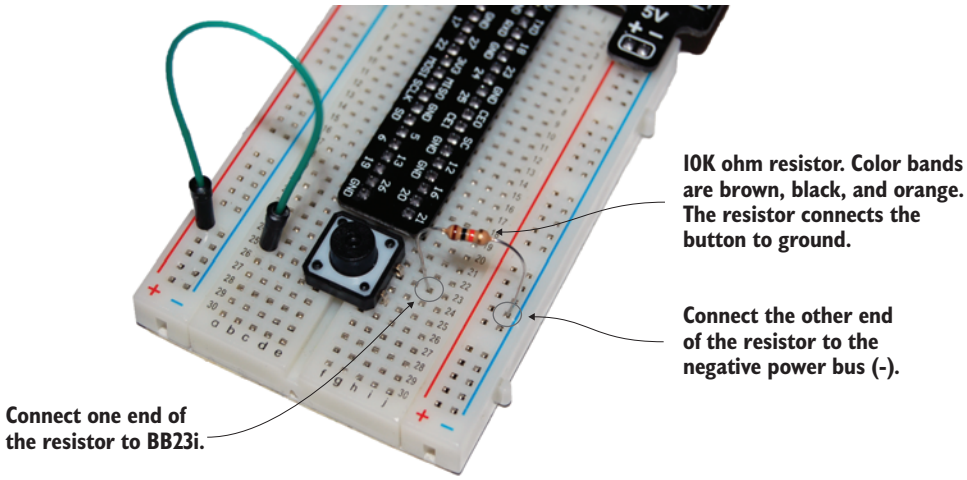


Figure 8.9 Add the resistor. Make sure its ends are pushed down into the breadboard holes.

electricity will flow either way through a resistor, so it doesn't matter which way you place it.

You're connecting the resistor from the top of the button to the negative power bus (-). This is the set of holes with a blue stripe next to it running along the edge of the breadboard.

Insert one end of the resistor into *BB23i* and the other end into the *negative power bus (-)*. You can choose any hole along the blue line. Once the resistor is added, you'll have something that looks like figure 8.9. Now you're ready for the final step.

STEP 4. ADD THE JUMPER TO A GPIO PIN.

A small amount of electricity needs to reach a GPIO pin (you'll use GP06), so you need a jumper wire from the top of the button to a hole next to the GPIO pin. To make this connection, add a jumper from *BB23a* to *BB16a*. Figure 8.10 shows how it looks.

When the button is pressed, a small amount of electricity will flow to GP06 and through the resistor to ground. Nothing happens yet, but next you'll write a Python program to detect that electricity and play some sounds.

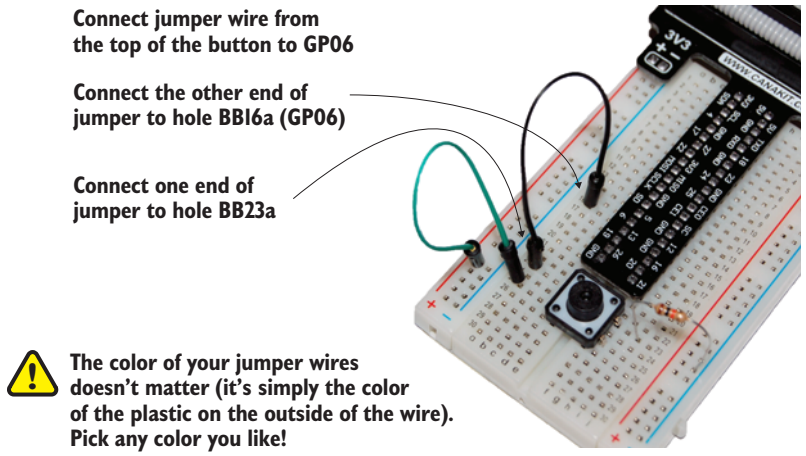


Figure 8.10 The jumper connects the top of the button to GP06. Later, you'll set your Pi to listen for electrical input on this GPIO pin.

Adding the second button

Let's add a second button to the board. Figure 8.11 shows what it will look like when it's done.

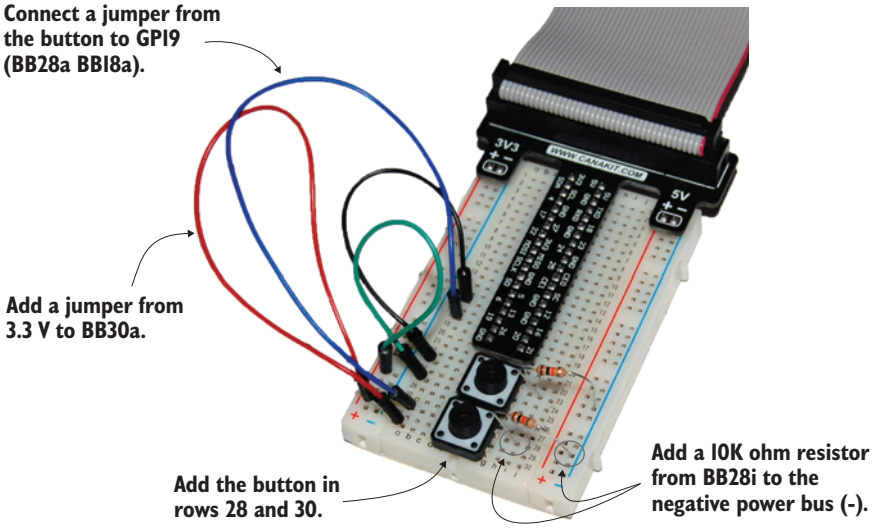


Figure 8.11 Add the second pushbutton just below the first one. The wiring is the same, but you'll connect it to GP19 (GPIO pin 19). Any available GPIO pin will work, but remember that your code will have to reflect the GPIO pins you select.

To add another button, you'll create the same circuit but place the button in rows 28 and 30 on your breadboards. You'll wire the button to GP19.

STEP 1. ADD THE MINI PUSHBUTTON.

Insert the button so that the top legs are in *BB28d* and *BB28g* and the bottom legs are in *BB30d* and *BB30g*.

STEP 2. CONNECT A JUMPER WIRE FROM 3.3 VOLTS TO THE BUTTON.

You need to connect power from the positive power bus to the bottom of the button. The power rail is the line of holes with a red line running next to it. Insert a jumper from anywhere along the *positive power bus (+)* to *BB30a*.

STEP 3. ADD THE 10K OHM RESISTOR.

To prevent too much electricity from flowing when the button is pressed, you need to add a resistor. As before, you'll add a 10K ohm resistor (color bands are brown, black, and orange) to connect the top of the button to the negative power bus (-).

Insert one end of the resistor into *BB28i* and the other end into the *negative power bus (-)*. Any hole along the blue line will work.

STEP 4. ADD THE JUMPER TO A GPIO PIN.

Finally, when the button is pressed, you need electricity to flow to a GPIO pin. For the second button, you're using GP19. Connect a jumper wire from *BB28a* to *BB18a* (GP19).

Terrific! The second button is connected, and you've completed the button circuit. Let's call the first button Button 1. It's wired to GP06. The second button, Button 2, is wired to GP19. Now that everything is wired up, let's write code for it!

Software: the DJ Raspi program

Your project is to turn your Pi into an awesome music player that is controlled by buttons. Here's how it will work:

- Pressing Button 1 makes the Pi play random music clips.
- Pressing Button 2 makes the Pi play random vocal (singing) sounds.

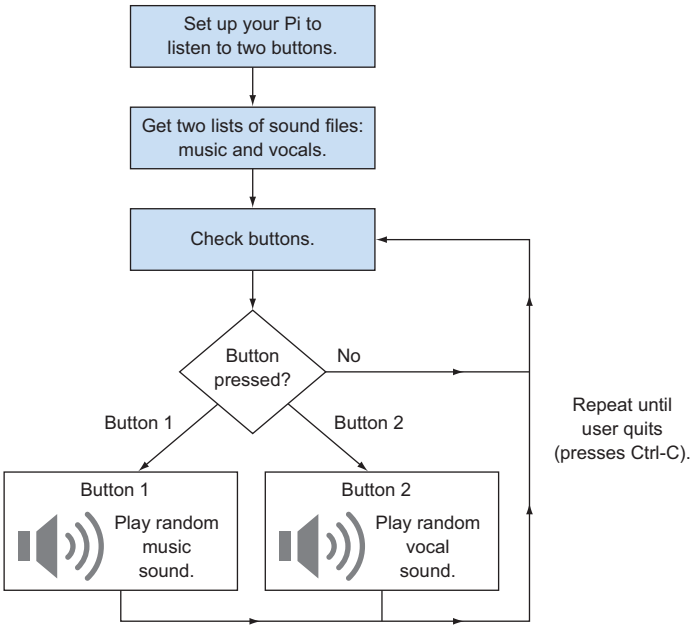


Figure 8.12 A flow diagram showing how the DJ Raspi program should work. The program must gather a list of sounds at the beginning and then check whether the buttons are pressed. The buttons will be checked over and over again.

You'll need one of the following to hear the sounds:

- Headphones
- Powered computer speakers
- Your Pi connected via HDMI to a TV with built-in speakers

Let's think through how this program will work. Figure 8.12 shows a quick diagram of the logic.

Let's write the code in this order:

1. Set up your Pi to listen to input coming from the buttons.
2. Gather a list of music and vocal sounds.
3. Program a loop to check the buttons. If they're pressed, then play random sounds.

You'll try to use functions along the way to simplify your code.

Let's begin! Open IDLE by choosing Python 3 under Menu > Programming. In the Python Shell, start a new program by pressing Ctrl-N or selecting File > New Window.

Setting up the Pi: initializing the buttons

In the IDLE text editor, you'll start by loading the Python libraries you'll need to use. You'll also set up a couple of the Pi's GPIO ports to listen for electrical signals coming in from the buttons being pressed. In the flow diagram, this is the first step of initializing the buttons (see figure 8.13).

When you set up the GPIO ports, you use `GPIO.IN` to tell the Pi that you plan to use that port as an input. To prepare your Pi for input to the

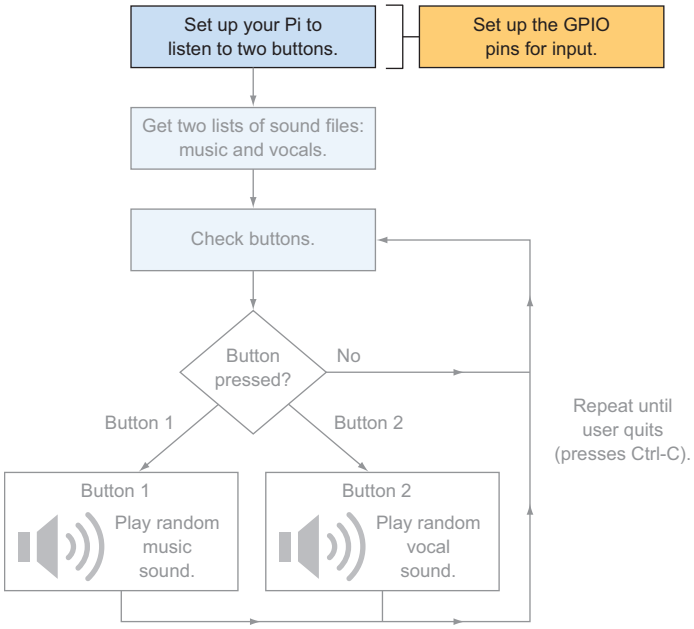


Figure 8.13 The first step is to set up the buttons as inputs. This will mean your Pi is ready to check whether it's detecting any voltage coming in, which will happen when a button is pressed.

GPIO pins, you need to tell it which pins you plan to use. Based on the circuit, you're using these pins as inputs:

- GP06 for Button 1
- GP19 for Button 2

The following listing shows how you can use the `GPIO.setup` command to set a GPIO pin to input.

Listing 8.1 Setting up GPIO pins for input

```
# DJ Raspi
# Ryan Heitz

# importing the libraries you need
import RPi.GPIO as GPIO
import time
import random
import os

# Variables for the button GPIO input pins
button_pin1 = 6
button_pin2 = 19

#Tell the Pi we want to use a breakout board
GPIO.setmode(GPIO.BCM)

# Set up GPIO pins as input pins (detect electrical signals coming in)
GPIO.setup(button_pin1,GPIO.IN)
GPIO.setup(button_pin2,GPIO.IN)
```

Import the os library that lets you execute a Raspbian command.

Store the value of the GPIO pins.

Set up the pins for input (notice you use GPIO.IN).

You may notice that you import a new `os` module. We'll talk about why you need that in the next section when you gather your lists of sound files.

Getting a list of sounds

Lists are everywhere around you. You make lists of things you need to do, gifts to buy, places you want to visit, and favorite things, such as your top-10 movies or books.

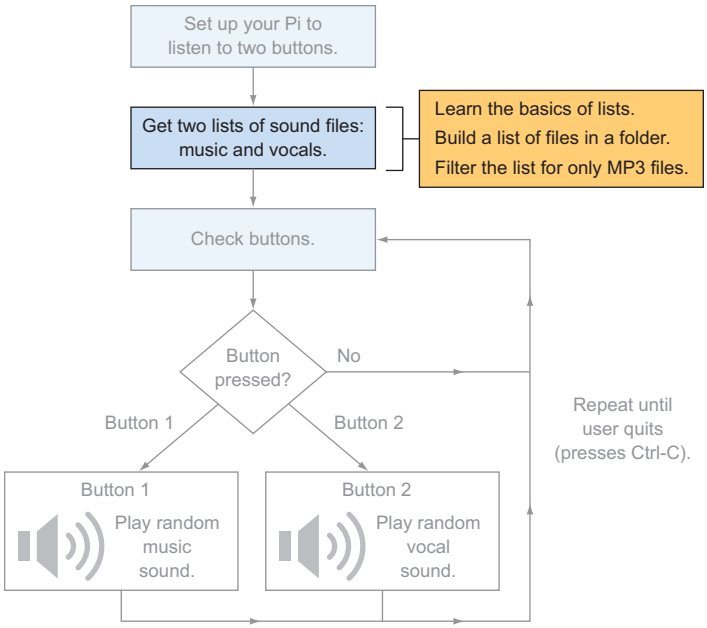


Figure 8.14 The next step of the DJ Raspi program gets a list of sound files. Later, you'll add the part that uses the button to trigger playing random sounds from the lists.

Your DJ Raspi needs a *list* of sound files: one for music clips (or loops) and one for vocals. Based on the design, you need to get a list of files from a folder on your Pi, and then you need to select a random sound file from the list and play it (see figure 8.14).

In Python, you can create lists or groups of things easily. Let's look at some examples.

Let's create a list of basketball player names. Open IDLE to the Python 3.x Shell by choosing Python 3 under Menu > Programming. In the Python Shell, make a list:

```
>>> basketball_players = ["Kevin Durant", "LeBron James", "Chris Paul",
➥ "John Wall"]
```

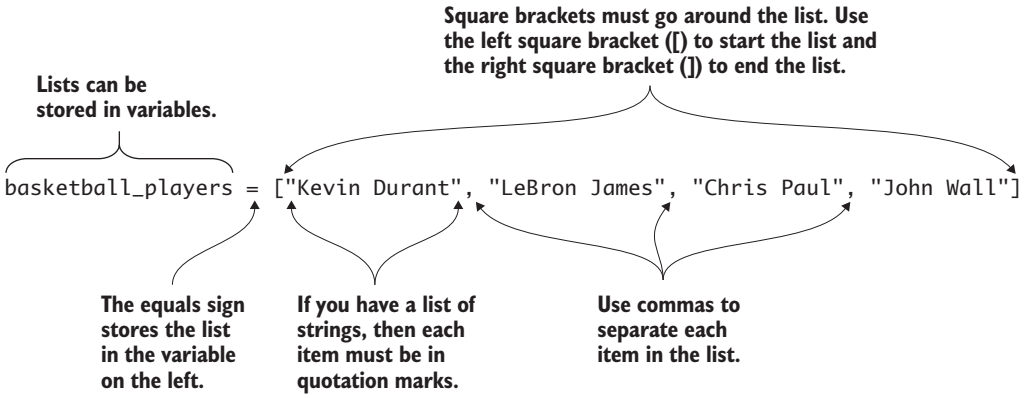


Figure 8.15 You make lists by using square brackets to enclose a set of things. Each thing in the list should be separated with a comma. Python will even let you make lists that combine different types of data, like strings and integers.

Print out the list like this, and you'll see what's inside:

```
>>> print(basketball_players)
['Kevin Durant', 'LeBron James', 'Chris Paul', 'John Wall']
```

To make a list of the items, put them in a set of square brackets (`[]`) and separate each item with a comma (see figure 8.15). For lists of strings, each item in the list has to have quotation marks around it. Pretty simple! That's the Python way.

Try creating a list called `favorite_numbers`, like so:

```
>>> favorite_numbers = [22, 27, 49, 121, 2, 25]
```

Display the contents of the list using `print`:

```
>>> print(favorite_numbers)
[22, 27, 49, 121, 2, 25]
```

> NOTE When making a list of numbers, you don't use any quotation marks.

Enjoy making lists of some of your favorite things!

More things you can do with lists

There are lots of things you can do with lists! Let's try a few.

You make a list longer by adding more items to it. To do this, use the `append` method. Let's add the name Stephen Curry to the list of `basketball_players`. Here is how you can use `append` to do that:

```
>>> basketball_players.append("Stephen Curry")
```

Use `print` to see the result:

```
>>> print(basketball_players)
['Kevin Durant', 'LeBron James', 'Chris Paul', 'John Wall', 'Stephen Curry']
```

Excellent! To remove an item from a list you can use the `remove` method. If you wanted to take John Wall out of the list, write

```
>>> basketball_players.remove("John Wall")
```

Print the list again to see if it worked:

```
>>> print(basketball_players)
['Kevin Durant', 'LeBron James', 'Chris Paul', 'Stephen Curry']
```

Wonderful! If you need to put a list in order alphabetically or from lowest to highest, you can use the `sort` method like so:

```
>>> favorite_numbers.sort()
```

Check that it worked by printing the list to the screen:

```
>>> print(favorite_numbers)
[2, 22, 25, 27, 49, 121]
```

The numbers are all sorted! This works on lists made of strings as well. If you sort the list of `basketball_players`, it puts them in alphabetical order based on the first letter of each string. Python has many built-in methods for lists.

Check the online Python documentation[a] for more things you can do with lists. Then sit back and enjoy thinking about all you can do with them in your future programs.

---

[a] Go to the Python website for more information on things you can do with lists: https://docs.python.org/3.4/tutorial/datastructures.html.

For your DJ Raspi, let's see how to

- Get the value of an item stored in a list.
- Get the length of a list.

Getting a value of an item stored in a list

Let's start with a fresh list of basketball players:

```
basketball_players = ["Kevin Durant", "LeBron James", "Chris Paul",
➥ "Stephen Curry"]
```

As you've seen, lists store information. What you might not know is that each spot in a list is given a number called the *index*. The index of the first item in the list is zero (0). The second item has an index of 1. The third item's index is 2, and so on. To get the third item in the `basketball_players` list, you'd type

```
>>> print(basketball_players[2])
Chris Paul
```

If you want to search a list and have Python tell you the index of where an item first appears in the list, you use the `index` method:

```
>>> basketball_players.index("Kevin Durant")
0
>>> basketball_players.index("Stephen Curry")
3
```

If the item isn't in the list, Python will give you an error saying so:

```
>>> basketball_players.index("Me")
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    basketball_players.index("Me")
ValueError: 'Me' is not in list
```

> NOTE Remember that the index for lists starts counting at 0, not 1! For example, `basketball_players[1]` gives you `"Lebron James"`, the second item in the list.

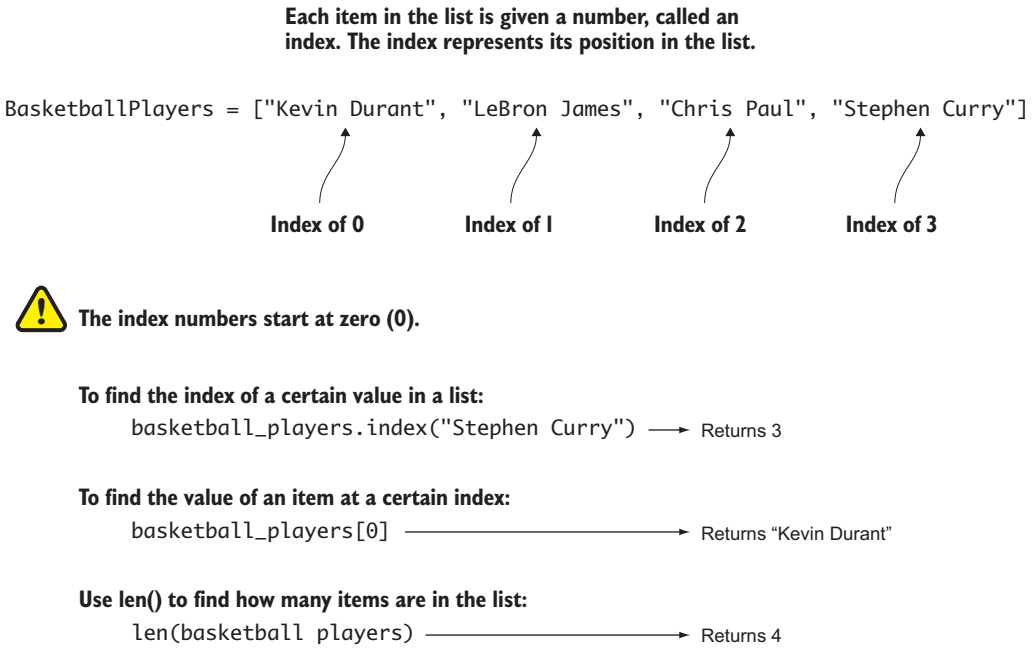


Figure 8.16 Sets of things can be stored in lists. You can retrieve items from the list using the index, which represents the position of an item in the list. The index of a list starts at 0.

Figure 8.16 shows examples of the indexes for a list and how you can get a specific item in a list.

Getting the length of a list

Finally, there are times when you've loaded information into a list and you need a way to check how long the list is. Use the `len()` function to do that:

```
>>> yummy_snacks = ["chips", "popcorn", "donuts", "cheese",
  "pretzels", "spam"]
>>> print(len(yummy_snacks))
6
```

Great job—you know the basics of lists. Now let's see how you can create lists of MP3s.

Building a list of sound files with the os library

To make the DJ Raspi project work, you need to

1. Grab two lists of sound files from folders on your Pi.
2. Make OMXPlayer play sound files from Python as part of the DJ Raspi program.

Let's learn how.

The Pi has both these abilities through a Python module called the `os` module (OS stands for *operating system*). With it, you can run operating system commands (things you can type in the Terminal window) from your Python programs. This is fantastic, because it means you can get lists of files and also call OMXPlayer to play a certain file—exactly what you need!

GETTING A LIST OF FILES FROM A FOLDER: USING LISTDIR()

Your Pi has some sound files on it already, as you saw in section 8.1. You'll use the files in these two folders:

- /usr/share/scratch/Media/Sounds/Music Loops/
- /usr/share/scratch/Media/Sounds/Vocals/

The `os` library provides a built-in function, `os.listdir(some_path)`, to get a list of files at `some_path`. To get a list of Scratch music loops and vocals, use these commands:

```
# Folders with sound files
path_music = "/usr/share/scratch/Media/Sounds/Music Loops/"
path_vocals = "/usr/share/scratch/Media/Sounds/Vocals/"

# Creating two lists with the files in the folders
sounds_music = os.listdir(path_music)
sounds_vocals = os.listdir(path_vocals)
```

If you print the lists, you'll have something that looks like this:

```
print(sounds_music)
['Cave.mp3', 'Techno.mp3', 'HipHop.mp3', 'Triumph.mp3', 'Medieval2.mp3',
➥ 'HumanBeatbox2.mp3', 'DripDrop.mp3', 'Xylo3.mp3', 'GuitarChords1.mp3',
```

```
➥ 'DrumSet2.mp3', 'Xylo2.mp3', 'DrumSet1.mp3', 'Garden.mp3',
➥ 'GuitarChords2.mp3', 'Jungle.mp3', 'Xylo1.mp3', 'Eggs.mp3',
➥ HumanBeatbox1.mp3', 'Drum.mp3', 'DrumMachine.mp3', 'Techno2.mp3',
➥ 'Medieval1.mp3', 'xylo4.mp3']

print(sounds_vocals)
['Oooo-badada.mp3', 'Singer1.wav', 'BeatBox1.wav', 'Ya.wav',
➥ 'BeatBox2.wav', 'Come-and-play.mp3', 'Hey-yay-hey.mp3',
➥ 'Doy-doy-doy.mp3', 'Singer2.wav', 'Got-inspiration.mp3',
➥ 'Join-you.mp3', 'Sing-me-a-song.mp3']
```

Wow—you have nice-looking lists! But wait: it looks like `sounds_vocals` has WAV (.wav) files and MP3s. Let's filter out the WAVs so you only have MP3s.

FILTERING FOR ONLY MP3S

To filter a list, you can use Python's *list-comprehension* feature. List comprehension is a quick way of creating lists. When you use it, you can include certain conditions or operations that are applied to the items in the list, such as making sure all the files in the list end with .mp3. Let's look at how you can use list comprehension to create a new list from your old list, but only keep the files in the list that end with .mp3:

```
sounds_music = [sound for sound in sounds_music if
  sound.endswith('.mp3')]
sounds_vocals = [sound for sound in sounds_vocals if sound.endswith
➥ ('.mp3')]
```

The list comprehension has a `for` loop inside it. In this case, Python is looping through the list of sound files in your original list of sounds. For each item in the list, Python only adds it to the new sounds list if it matches the condition of being a file ending with .mp3.

Playing a sound when a button is pressed

Next in your plan is to write the code that will play a random sound from your lists when a button is pressed. You'll need this to be in a loop

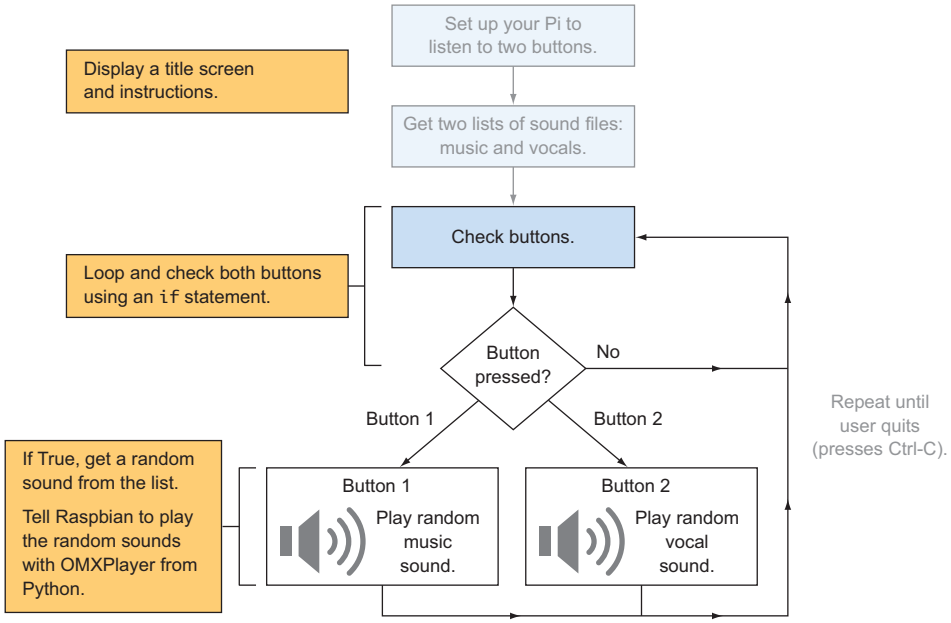


Figure 8.17 The main part of the DJ Raspi program is the loop to check the buttons. You'll use a `while` loop to check the buttons over and over again. If one of them is pressed, you'll tell Raspbian to play a random sound using OMXPlayer.

so the buttons are repeatedly checked to see whether they're being pressed (see figure 8.17). Let's start by creating the game's title and creating the main game loop.

LOOP TO CHECK THE BUTTONS

First let's add some code to display a title screen and the DJ Raspi instructions. Feel free to make the title screen fancier!

Listing 8.2 DJ Raspi title screen

```
# Clear the screen
os.system("clear")
```

The clear command makes the Terminal a blank, black window.

```
#Display a title screen
title = """
    DJ RASPI!!!
    Press Button 1 for Music Sounds
    Press Button 2 for Vocal Sounds
    Press Ctrl + C to exit
"""
print(title)
```

Use triple quotation marks (""") to create a string literal for the title.

Now let's write the code to loop over and over again to check whether either button is being pressed. When a button is pressed, the GPIO pin will give you a response of True, and you can then call a function to play a random MP3.

Listing 8.3 DJ Raspi game loop

```
# Start an infinite loop (must use Ctrl-C to stop it)
while True:
    if GPIO.input(button_pin1):
        print("You pressed #1!")
        play_random_sound(path_music, sounds_music)
        time.sleep(.1)
    if GPIO.input(button_pin2):
        print("You pressed #2!")
        play_random_sound(path_vocals, sounds_vocals)
        time.sleep(.1)
    time.sleep(.1)
```

Play a random sound using the function you wrote.

If the GPIO detects input, this is True, and the sound is played.

Wait a small amount of time.

The code repeatedly checks whether Button 1 or Button 2 is pressed. If Button 1 is pressed, the code plays a random music sound. If Button 2 is pressed, the code plays a random vocals (singing) sound. If neither is pressed, the code loops around and checks them again. The loop never ends, so you'll need to press Ctrl-C to exit the program.

PLAYING SOUNDS: USING OPERATING SYSTEM COMMANDS FROM PYTHON

You're ready to play your sounds! The `os` module will let you run operating system commands (ones you normally run using Terminal). To play the first sound in the `sounds_music` list, you could write

```
os.system("omxplayer -o local '" + path_music + sounds_music[0] + " &")
```

Later in this chapter, we'll explain why the end of that command has an ampersand (&). The result of this command would be the same as typing this at the Raspbian command line:

```
omxplayer -o local "/usr/share/scratch/Media/Sounds/Music Loops/
  Cave.mp3"
```

> NOTE Remember, if you're outputting the sound to HDMI (if your TV has speakers), you need to change `-o local` to `-o hdmi`.

Excellent! Let's review what you've learned so far:

- Your Pi can play sounds that are in MP3 format using OMXPlayer.
- Python can store sets of things as lists.
- The Python `os` library has a function called `listdir(path)` that can give you a list of sounds in a folder.
- Python's `os` library has an `os.system(command)` function that can run operating system commands from Python, such as playing sounds with OMXPlayer.

Functions!

Let's think about how you can write the functions for DJ Raspi. You'll want to create two functions:

- `get_MP3_sounds`—This function will get a list of sounds ending in .mp3 from a specified folder. You'll tell the function (pass it a parameter) the name of the folder where you want to get the MP3 sound files. The function will return a list of sounds.
- `play_random_sound`—This function will take a list of sounds, pick a random number, and then use `os.system` to tell Raspbian to play the sound with OMXPlayer.

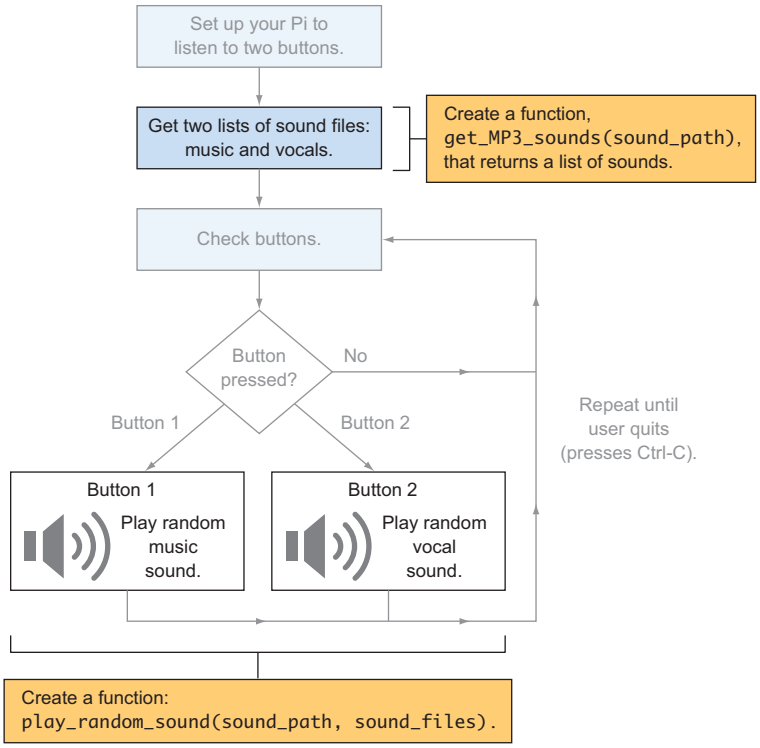


Figure 8.18 There are two places where you can create functions so you can reuse code. One function creates a list of sound files, and the other function plays a random sound when a button is pressed.

Figure 8.18 shows where these functions fit into the flow diagram.

Why not make this a single function? One reason is that you only need to load a list of sound files once, near the beginning of the program. You play the sound files every time a button is pressed. Listing 8.4 shows the code for the `get_MP3_sounds` and `play_random_sound` functions.

> NOTE Remember to put these functions near the beginning of the program. They must be added before they're used for the first time.

At the end of this listing, you use (or call) `get_MP3_sounds` twice to get your lists of music and vocal sound files.

Listing 8.4 Functions for loading and playing sound files

```
# Returns a list of mp3 sound files for the path given
def get_MP3_sounds(sound_path):
    sound_filesound_files = os.listdir(sound_path)
    sound_filesound_files = [sound_file for sound_file in
   sound_filesound_files
                  if sound_file.endswith('.mp3')]
    return sound_filesound_files

# Plays a random sound from a list of mp3s for the path given
def play_random_sound(sound_path, sound_filesound_files):
   random_sound_index = random.randint(0,len(sound_filesound_files)-1)
   os.system("omxplayer -o local '" + sound_path +
             "/" + sound_filesound_files[random_sound_index] + "' &")

# Get the list of music loops and vocals (mp3s only)
sounds_music = get_MP3_sounds(path_music)
sounds_vocals = get_MP3_sounds(path_vocals)
```

Use the listdir function to get a list of files in a folder on your Pi.

Comprehension filters the list to only keep files ending in .mp3.

Use the randint and len functions to get a random sound number.

Play the sound with os.system.

You may have noticed that a few extra things are added to this line:

```
os.system("omxplayer -o local '" + sound_path +
          "/" + sound_filesound_files[random_sound_index] + "' &")
```

This line joins the command to run OMXPlayer with the path to your sound files (`sound_path`) and the random sound file you want to play (`sound_filesound_files[random_sound_index]`). At the end, you add an ampersand (`&`). The ampersand tells Raspbian to run the command in the background. This is so you can quickly press one button and then the other.

Doing multiple things at once: meet the ampersand (&)

When you play sounds, you want to be able to press the buttons quickly, like a DJ, and have the sounds overlap to create interesting music. Normally, your Pi would play one sound, and, when it was finished playing, let you play another. Not what you want. Here is an example of playing two sounds. You can't run the second command until the first one is finished:

```
omxplayer /usr/share/scratch/Media/Sounds/Vocals/Oooo-badada.mp3
omxplayer /usr/share/scratch/Media/Sounds/Vocals/Hey-yay-hey.mp3
```

(continued)

Luckily, your Pi can do a few things at once. You might have several different windows open at the same time. Each window is connected to some underlying code or set of instructions running on your Pi. These underlying sets of code are called *processes* or *threads*. Raspbian, like other modern operating systems, manages these processes and assigns each one its own unique ID number. An ampersand (&) placed at the end of a command tells your Pi to run the command as another process in the background alongside any other processes.

Try these two commands again, but this time with ampersands:

```
omxplayer /usr/share/scratch/Media/Sounds/Vocals/Oooo-badada.mp3 &
omxplayer /usr/share/scratch/Media/Sounds/Vocals/Hey-yay-hey.mp3 &
```

Notice the ampersand (&) at the end of each command. In this way, the Pi will play your sound, but the code won't make your Pi wait for the sound to finish before doing something else. Adding an ampersand at the end of the OMXPlayer command makes the button play each sound in the background. Remove the ampersand to play one sound at a time. Because this is a feature of the OS, the concept of using ampersands to execute commands as their own unique processes applies to other Linux commands that you know already or will learn.

Great! You're ready to test your project!

Testing: your first gig as DJ Raspi

Save the code as DJRaspi.py, and try running it. Select Run > Run Module (or press F5) from the IDLE text editor to run your program. If you have an older version of Raspbian (prior to October 2015), programs that use the GPIO pins must be run from the Raspbian command prompt as the superuser (or root). Open Terminal, and enter the following command:

```
pi@raspberrypi ~ $ sudo python3 DJRaspi.py
```

You should see the title screen display. Test it by pressing the buttons to see if they work.

> NOTE Remember that any program that uses GPIO pins must be run from the Raspbian command prompt as the superuser (or root).

Believe it or not, it's rare for a program to work perfectly the first time. If it doesn't, read through the following "Troubleshooting"

section and review your circuit and program to try to figure out how to get it working.

Troubleshooting

If sounds aren't playing when you press the buttons, here are some things you can check:

- Check the circuit on the breadboard. Is the ribbon cable connected properly, with the first wire connected toward the edge of your Pi, away from the USB ports?
- Double-check that the jumpers, buttons, and resistors are in the right holes and pressed all the way into the breadboard.
- Does your program print "You pressed #1!" and "You pressed #2!"? If it does, you know your circuit is working, and either it's an issue with the code to load the sound files or your speakers or headphones aren't working. Try running one of the following commands from Terminal to check whether the speakers are working.

  For headphones or speakers plugged into the 3.5 mm audio output:

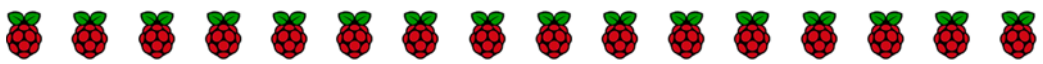
```
omxplayer -o local /usr/share/scratch/Media/Sounds/Vocals/
➥ Oooo-badada.mp3
```

  For TV speakers connected by HDMI:

```
omxplayer -o hdmi /usr/share/scratch/Media/Sounds/Vocals/
➥ Oooo-badada.mp3
```

- Look through your Python program for errors. Try adding some `print` statements to your functions to make sure they're getting the list of MP3 files properly.

If you've enjoyed creating DJ Raspi, check out the button challenges.



Challenges

Try some of these button activities for extra fun!

Double button press surprise

Give your program a surprise button combination. See if you can make pressing both buttons at once play a new set of sound effects.

Hint: When you want something to happen only if both conditions are True, you can use the ampersand (`&`). The `if` statement will only be True if both the first *and* second conditions are True. It looks like this:

```
if GPIO.input(button_pin1) & GPIO.input(button_pin2):
    print("Both buttons are pressed!")
```

Here is the path to some Scratch sound effects on your Pi:

```
path_effects = "/usr/share/scratch/Media/Sounds/Effects/"
```

Yoda Magic 8 Ball

There is a great classic toy called the Magic 8 Ball. It's a ball that displays an answer to a question when you shake it. Ask it a question, and you'll get some truly magical advice. The Magic 8 Ball has 20 different answers, ranging from "It is certain" to "My sources say no."

> NOTE I don't recommend using the Magic 8 Ball to advise you on major life matters!

Your challenge is to make a Magic 8 Ball program:

- Ask a question aloud, and then press a button.
- Pressing the button makes your Pi select a random Yoda clip from a folder and play it.

To get started, find some short sound files of Yoda sayings. One place to find them is on soundboard.com. Search for "yoda" to see if you can locate some good clips (you'll need to create an account to download them for your personal use).

Bonus: Try to give your Pi a handy button that plays Monty Python sound clips whenever you press it.

Continuing to explore

Now that you've given you Pi a new sense of touch, you'll only need to change a few lines of code to make many other projects, such as these:

- An interactive display that shows different digital photographs each time a button is pressed
- Your own Raspberry Pi movie player that plays clips or movies at the press of a button
- An MP3 music player that shuffles through your favorite songs

You can also expand past buttons to sensors, such as passive infrared (PIR) sensors or cameras. For example, PIR sensors detect motion near the sensor. These are great for creating a Pi security system or something that scares people when they come to your door. Maybe you want to trigger a movie to make a frightening zombie head appear or generate a blood-curdling scream. Your only limit is your imagination and mischievous thoughts.

Summary

In this chapter, you learned that

- A Pi can sense the environment around it using the input capability of the GPIO pins. This creates incredible possibilities to make the Pi have human or even superhuman senses.
- Python lists make it easy to store and retrieve sets of things like numbers, sound files, images, and videos.
- Buttons act as simple switches that send a small amount of electricity to your Pi's GPIO pins, which it can detect. You have nothing to fear in wiring up buttons or other sensors!
- Python programs can run Raspbian commands using the `os` library. This opens lots of possibilities for your programs, from playing

music to showing or taking videos, displaying or taking pictures, and accessing information from websites.

You've completed a great adventure in learning Python programming and how to use your Raspberry Pi. But there is much more excitement ahead of you. Check out appendix D for even more ideas of projects you can do with your Pi. With your Raspberry Pi, knowledge of Python, and a bit of fearlessness, the possibilities are endless!

Appendix A

Raspberry Pi troubleshooting

In this appendix, you'll learn how to solve common issues when setting up a Raspberry Pi. We'll cover common Pi startup (or boot) issues, including how to fix an issue with an SD card for your Pi or set up a new SD card.

Making sure your Pi has power

Sometimes a Pi won't start up. Before you try something drastic like creating a new SD card for your Pi, check the Pi's power:

- When you plug in your Pi, does the Pi's red power light come on?

  Look for a small, red light (LED) on your Pi board. All Pis have them, but you may need to take off your case if you can't see the Pi board. The red power light tells you that your Pi is receiving power. It should come on when you plug in your Pi and stay on the whole time you're using it. If it doesn't come on, that means your Pi isn't receiving power. Check that the power supply is plugged in. If you're using a power strip, check that it's turned on. Sadly, some power supplies are poorly made. Get a new power supply if it's a power issue.

- Next to the red light, does the green activity light (LED) flash a lot when you plug in your Pi?

The flashing is a sign that your Pi is doing some work. The green activity light should turn on and off, flashing quickly at times, as your Pi boots up. When it's done starting up, the green light will turn off and only come on when your Pi is actively doing something like opening a game or Python. If the green light comes on and stays on, but nothing is displayed to the screen, it's likely an issue with your SD card. Jump ahead to section A.4 to learn how to create a new SD card.

If you suspect a power issue, purchase a new power supply and try it out. Providing sufficient electrical power (2 amps) at the correct voltage (5 volts) is important for a Pi to work correctly.

Checking the connection to your TV or monitor

If the red light is staying on and the green light turns on and off after you plug in your Pi, but you don't see any image on your TV or monitor, it's time to check the connection to your screen. Here are a few things to investigate, depending on the type of TV or monitor you're using.

If you're connecting an HDMI cable from your Pi to your TV or monitor, try these things:

- Check that the TV or monitor is turned on.
- Check that the TV or monitor is set to the correct input. They typically have multiple inputs, and you must press an input button to select the proper one. Otherwise, the screen will display nothing or a message saying no input is detected.
- If you have an extra HDMI cable, try using it to see if it's an issue with the cable.

If your setup requires that you use an adapter to connect your Pi to your TV or monitor, then you may need to make sure your adapter works or is the correct type. There are two common types of adapters:

- *HDMI-to-DVI adapter*—This adapter is used to connect the Raspberry Pi's HDMI cable to a monitor with a digital visual interface (DVI) port. Sometimes you might purchase a bad adapter that doesn't work. If you can, try connecting your Pi to another TV or

monitor that uses HDMI to check whether it's an issue with the adapter. Again, if the red light comes on and the green light flashes, but you don't see anything on your screen, it's likely an issue with the monitor connection.

- *HDMI-to-VGA adapter*—Older monitors don't have HDMI or DVI ports and may only have a video graphics array (VGA) port. Your only option may be to buy an HDMI-to-VGA adapter to connect your Pi to the monitor. Not all HDMI-to-VGA adapters work. Your best bet is to buy one that is advertised to work with the Pi from a store that sells Raspberry Pis. If you aren't sure, try hooking up your Pi to another TV or monitor using only the HDMI cable to test whether that is your issue.

If, after all those steps, you don't see a picture, it's likely an issue with your SD card.

Pi starts booting up but then stops

Another issue you may see is that your Raspberry Pi starts booting up, you see a series of messages displayed on the screen, and then the messages stop but the Pi doesn't reach the Raspbian desktop or command line. If this is the case, it's likely that the SD card has been damaged. But another reason is that there could be something wrong with one of your GPIO pins.

If you're building circuits on a breadboard connected to your Pi (see the examples in chapters 6–8), the Pi may fail to boot all the way up if a wire is improperly connected. To see whether this is the issue, disconnect the ribbon cable and breadboard from your Pi and try powering it up again. If the problem persists, it's likely an issue with the SD card.

Making your Pi a new SD card

Still not starting up? An issue with an SD card is a common reason. The SD card may stop working if the Pi is turned off when information is being stored (or written) on the card, or it may fail with age.

> NOTE When an SD card fails, you'll need to start over, and you'll lose any data or new applications installed on the card. In the future, you can create a backup of your SD card. Check out online forums to learn how to do this.

You have a couple options if you think this is the problem:

1. Clean and reset your SD card with the Raspberry Pi New Out of the Box Software (NOOBS).
2. Purchase a new card from one of the many online stores that sell Raspberry Pis. They cost around $10.

Let's go over how to clean and set up your SD card with NOOBS. To perform these steps, you need another computer, such as a Windows PC or a Mac.

Reformatting your SD card

Formatting is the process of setting up the memory storage so that information can be put on it. To reformat your SD card and set it up with a fresh version of NOOBS, do the following:

1. Using your other computer, download and install the SDFormatter software from the SD Association website: https://www.sdcard.org. On the website, look under Downloads, and download the appropriate version of SDFormatter for either Windows or Mac. Follow the install instructions to load the software on your computer.
2. Insert your SD card into the computer. Take note of which drive letter is assigned to the SD card after it's inserted: it may be E:, F:, or similar.

> NOTE For the Raspberry Pi Model 2 and B+, the SD card is a microSD card, so you'll need a microSD-to-SD card adapter in order for it to insert into the SD card slot on your computer. You can purchase such an adapter online.

3. Open SDFormatter, select the correct drive letter for your SD card, and then click Format (see figure A.1). It'll ask if you want to continue. Accept the warnings to reformat the SD card.

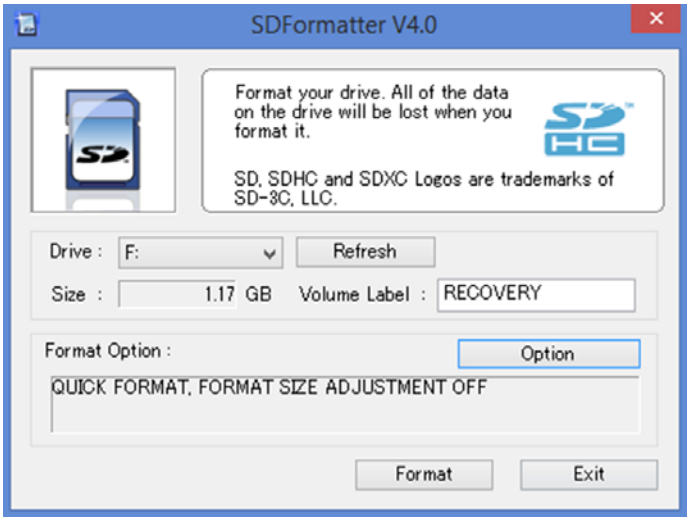


Figure A.1 Clean up (or reformat) your SD card using the SDFormatter software available from the SD Association website. In SDFormatter, select the drive letter for your SD card, and click Format to have your card wiped clean. This process means you're starting over—you lose anything saved or installed on your Pi—but sometimes that is the only option.

4. Go to the Raspberry Pi website at https://www.raspberrypi.org/downloads, and click the link to download the NOOBS zip file. It's a large file, so grab a snack while you're waiting for the download.
5. Extract the NOOBS zip file, and drag all the extracted files onto your SD card.
6. Once the files have been copied, take the SD card out of the computer and put it into your Raspberry Pi. With your keyboard, mouse, and monitor connected, plug in the Pi to see if it boots up. If it doesn't, it's probably time to purchase a new NOOBS SD card for your Pi from a local or online store that sells Raspberry Pis.

Problems not covered here

Not everything can be covered here, so get online! A large amount of troubleshooting information is posted on the Raspberry Pi forums. If

you're stuck, search the internet for "Raspberry Pi troubleshooting," and you'll find numerous resources. Although we're all special, it's rare to have an issue with the Pi that no one else has discovered. Chances are, many other people have had the same issue, so read the forums to benefit from all the knowledge that comes from the diverse community of Raspberry Pi users!

Appendix B

Raspberry Pi ports and legacy boards

In this appendix, you'll find information about some of the Raspberry Pi ports and connections that we didn't discuss in chapter 1. Our focus is on the Raspberry Pi 2 Model B. The connections and ports that we'll cover in more detail include the following:

- Wireless internet connections using a USB Wi-Fi adapter
- 3.5 mm audio/video port
- Camera Serial Interface (CSI) port
- Ethernet port
- TV or monitor connection options

In section B.2 of this appendix, we'll review key differences between the legacy Raspberry Pi 1 models as compared to the Raspberry Pi 2 Model B. We'll look more at these popular, but older models:

- Raspberry Pi 1 Model B rev 2 (released September 2012)
- Raspberry Pi 1 Model B+ (released July 2014)

Let's take a closer look at ports and connections.

Raspberry Pi ports

The Raspberry Pi has many different ports, and you can connect many different things to it. In chapter 1, we covered most of the common ones you'll use, but we'll talk about a few of the other ports here. For reference, figure B.1 shows the ports and their typical uses for the Raspberry Pi 2 Model B.

Now let's look in a little more detail at some of the ports and connections we didn't cover in chapter 1 or in later chapters.

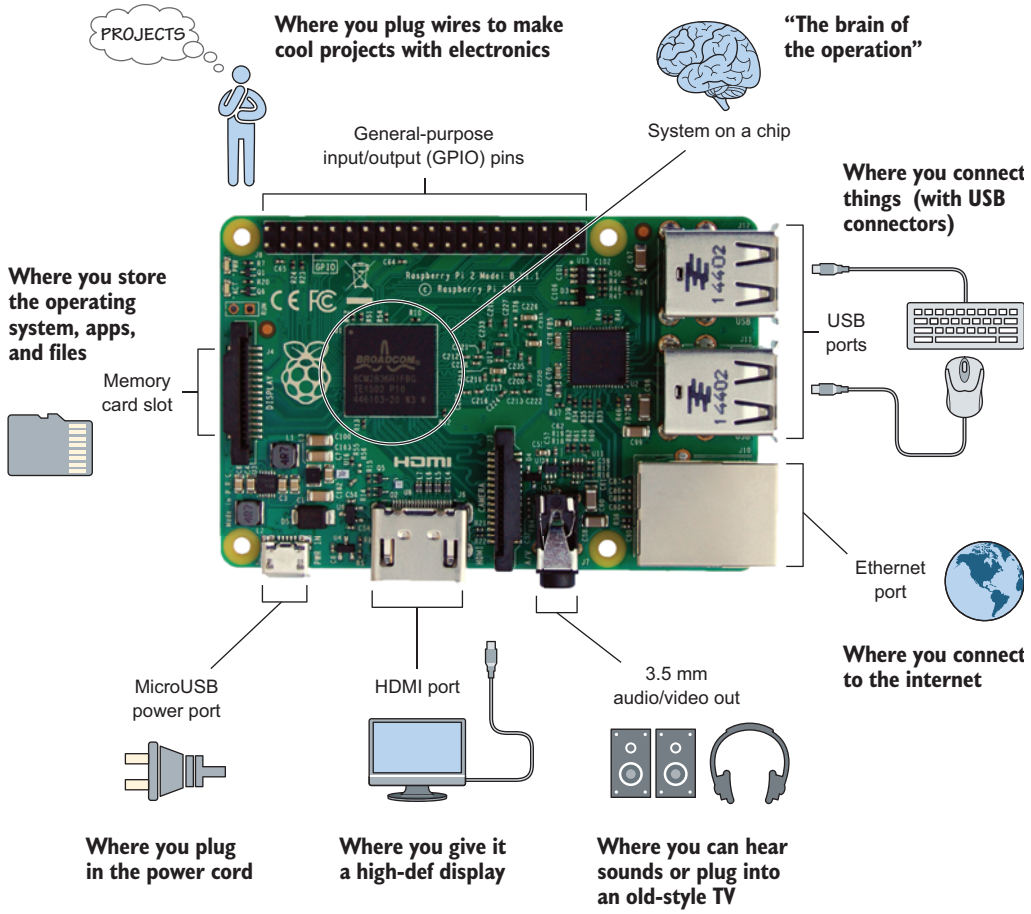


Figure B.1 The Raspberry Pi 2 Model B has many different input and output ports that allow you to connect a keyboard or mouse, monitor, and even high-definition cameras.

Connecting to a wireless network

A preferred way to connect to the internet is using a USB Wi-Fi adapter. Once connected, you can surf the web, download applications from the Pi Store, or remotely log in to your Pi from another computer. Most of us don't have our Pi set up near an Ethernet cable, so connecting wirelessly is the best and only option. Let's look at how you do it.

PLUGGING IN YOUR USB WI-FI ADAPTER

With your Raspberry Pi turned off, plug your USB Wi-Fi adapter into one of the USB ports. There are many different USB Wi-Fi adapters that will work for the Pi. Most kits come with one, but if you need to buy one, refer to the Raspberry Pi forums (see https://www.raspberrypi.org/forums/) to research those that are known to work. Stores that sell Raspberry Pis also tend to sell compatible USB Wi-Fi adapters.

CONFIGURING YOUR WI-FI CONNECTION

To connect to a Wi-Fi network for the first time, follow these steps:

In the top-right corner, click the network icon (looks like two small computers connected). You'll see a list of available Wi-Fi networks (see figure B-2).

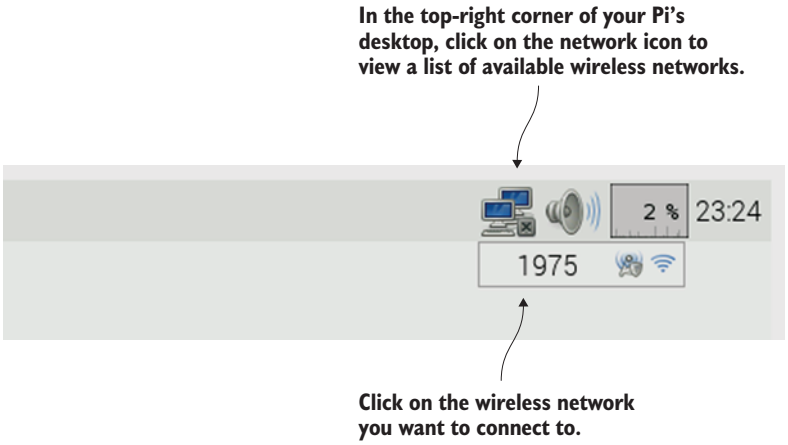


Figure B.2 The network connection icon is located near the top-right corner of the Raspbian desktop. Clicking on it allows you to view nearby wireless networks.

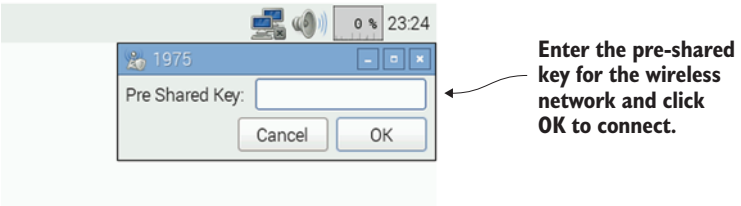


Figure B.3 Enter the pre-shared key for the network and click OK to connect to it.

Click on the Wi-Fi network name that you want to connect to.

Clicking on a Wi-Fi network name will make a small box appear. Enter the pre-shared key (also called the Wi-Fi password) and click OK to connect (see figure B.3). The network icon will change into a Wi-Fi icon showing the strength of the Wi-Fi signal.

Fantastic! Open a web browser, such as your Raspberry Pi's Epiphany web browser, and go to one of your favorite websites to enjoy your new Wi-Fi connection.

TROUBLESHOOTING

If you weren't able to connect, check that your pre-shared key was entered correctly. If it's correct and your web browser shows an error message saying "cannot resolve hostname," then your Pi may need to renew its IP address. The IP address is a unique series of numbers that a wireless router assigns to your Pi and other devices on the network. To renew your Pi's IP address, open the Terminal and enter these two commands:

```
pi@raspberrypi ~ $ sudo dhclient -v -r eth0
pi@raspberrypi ~ $ sudo dhclient -v eth0
```

If you still are unable to connect to the internet, check with the person who set up or manages the network to get help.

3.5 mm audio/video port

Whether it's Beethoven, Lady Gaga, or the creeper explosions in Minecraft, you'll want to listen to sounds on your Pi. Meet the 3.5 mm

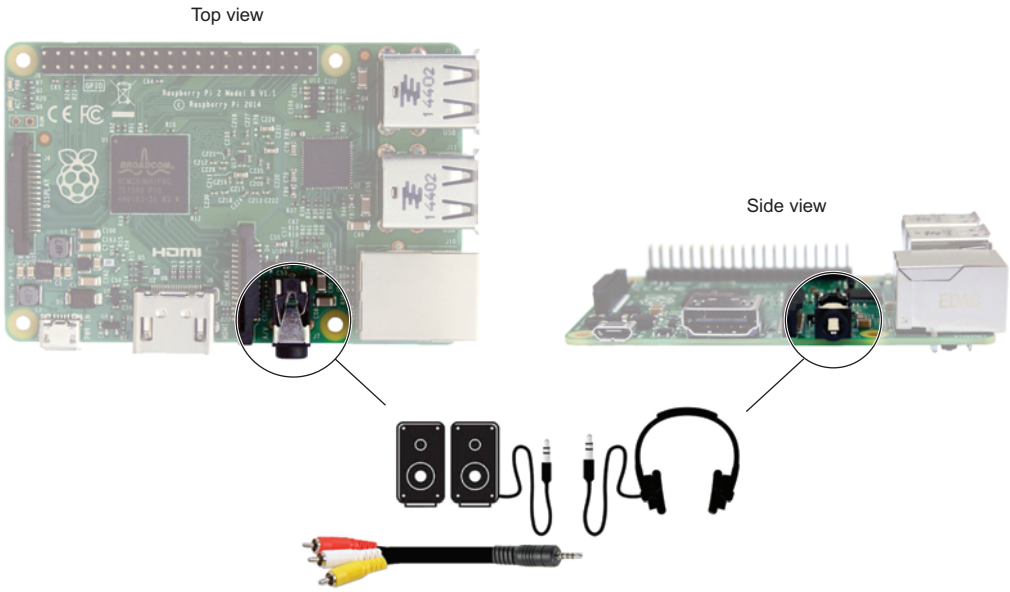


Figure B.4 The Raspberry Pi's 3.5 mm audio/video port is used to connect headphones or speakers for playing sounds. It can also function as a low-quality video output, if you purchase a 3.5 mm-to-RCA composite video adapter.

audio/video port (see figure B.4). This port is a black connector[1] with a round hole. It gets its name from the fact that the hole is 3.5 mm in diameter. Connect either headphones or a set of powered-computer speakers to listen to sounds from your Raspberry Pi.

> TIP If you connect computer speakers, use powered speakers, such as the type used with a desktop computer or iPod. The sounds that come out of the Raspberry Pi 3.5 mm audio/video port are only loud enough for a set of earphones or headphones. If you would like a roomful of people to hear your music, connect a set of powered speakers, which contain a built-in amplifier to boost the sound.

Starting with the Raspberry Pi 1 Model B+ and the Raspberry Pi 2 Model B, this port can also be used to output a video signal. The video

[1] In older versions of the Raspberry Pi, the port may be blue rather than black.

signal isn't high resolution like the HDMI port, but in a pinch, it's an option. The output video signal is composite or single-channel video, meaning all the video signal comes out in a single wire. It's what many of the older DVD players and video game consoles used at one time. You can purchase a cable that plugs into the port and at the other end has RCA connectors for plugging your Pi into an older TV.

Camera Serial Interface: connecting a camera

If you'd like to try time-lapse photography or set up a camera to take pictures of wild animals or your pet, you'll want to add a camera to your Pi. The best way to add a high-def digital camera to your Raspberry Pi is with the Raspberry Pi camera module. Created by the Raspberry Pi Foundation, it doesn't usually come with Pi kits, so you'll have to buy it separately. The module contains a 5 megapixel camera mounted on a circuit board and comes with a short ribbon cable (see figure B.5).

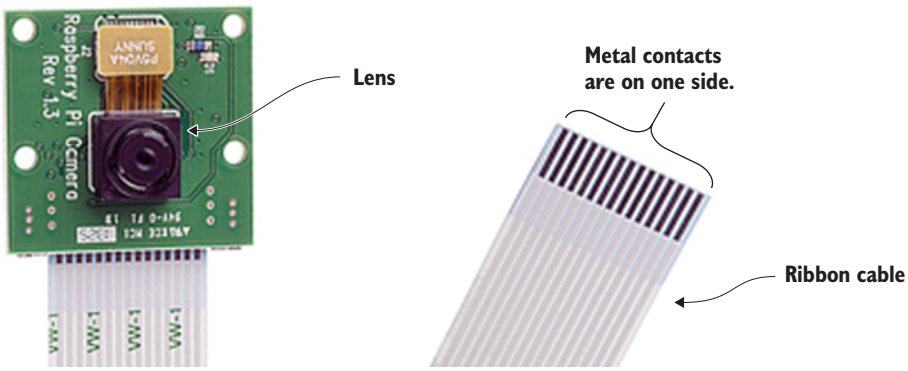


Figure B.5 The Raspberry Pi camera module was created by the Raspberry Pi Foundation to take high-def digital photographs and video. The camera attaches to the Pi using a ribbon cable that connects to the Camera Serial Interface port. The camera module can be programmed using Python and used for nature photography or creating your own home-surveillance system.

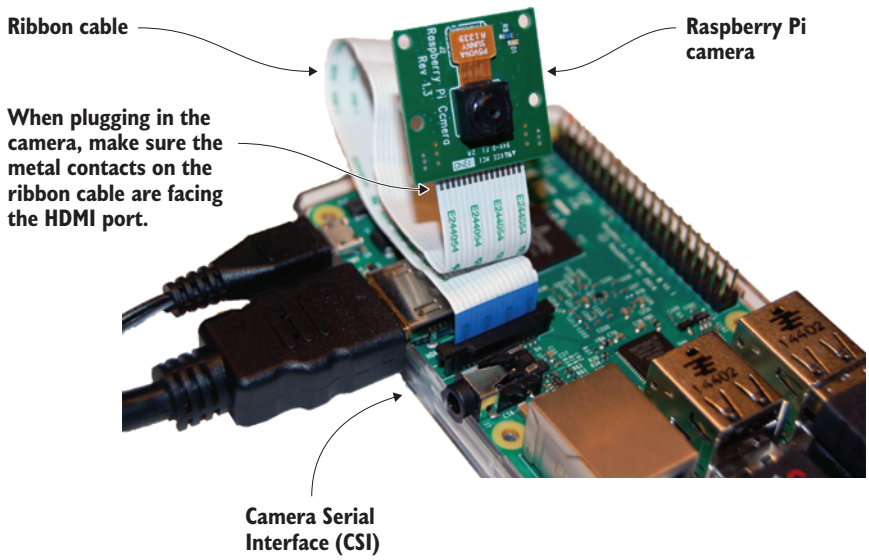


Figure B.6 The Raspberry Pi camera can connect to the CSI port, which is located between the HDMI port and the 3.5 mm audio/video port. To connect a camera, you need to lift up the black connector, insert the ribbon cable (metal contact toward the HDMI port), and push down on the black connector again.

The camera connects to your Pi's Camera Serial Interface (CSI) port (see figure B.6) and can take still photographs or high-def video. The module is able to connect easily to the Pi and record high-def video while consuming less processing power than using a USB camera with your Pi.

To connect the camera module, follow these steps:

1. Open the CSI connector on your Pi by lifting up on the top portion of the black plastic connector.
2. Insert the end of the ribbon cable into the CSI connector. The shiny metal contacts on the ribbon cable should face away from the Ethernet port and toward the HDMI port (see figure B.6).

3 Push the black plastic connector back down to close it, clamping the ribbon cable into the connector.

> TIP The Pi camera board comes with a short ribbon cable. If you need a longer one, you can find extension cables at online stores such as Adafruit.

Once the camera module is connected, you need to enable it. Open Terminal to enter Raspbian command-line mode. Enter the command to open the Raspberry Pi configuration menu:

```
pi@raspberrypi ~ $ sudo raspi-config
```

When the blue screen and Raspberry Pi configuration menu appear, select option 5: Enable Camera. Select Enable, and then select Finish on the main configuration menu. Your Pi will ask if you would like to reboot now; select Yes. When your Pi has rebooted, test out your camera by opening Terminal and typing

```
pi@raspberrypi ~ $ raspistill -t 3000 -o PiPhoto.jpg
```

This will turn on the camera and take a picture after 3 seconds. The `-t 3000` part tells the `raspistill` program the time to wait—in this case, it's set it to 3,000 milliseconds or 3 seconds. The image is saved to a file called PiPhoto.jpg. You can view the file by opening File Manager and looking in your pi\home folder. Check out online resources for more that you can do with your camera, including using PiCamera, a Python library for controlling the camera.

If you're thinking about taking videos or photos at night, there is an alternate version of the Pi camera module called the Pi NoIR (for near infrared) camera module. It uses the same CSI port and connects the same way. One difference is that you'll need to shine an infrared light source at the target you're filming. With the Pi NoIR, you're armed for some great new possibilities for nighttime mischief with your Pi.

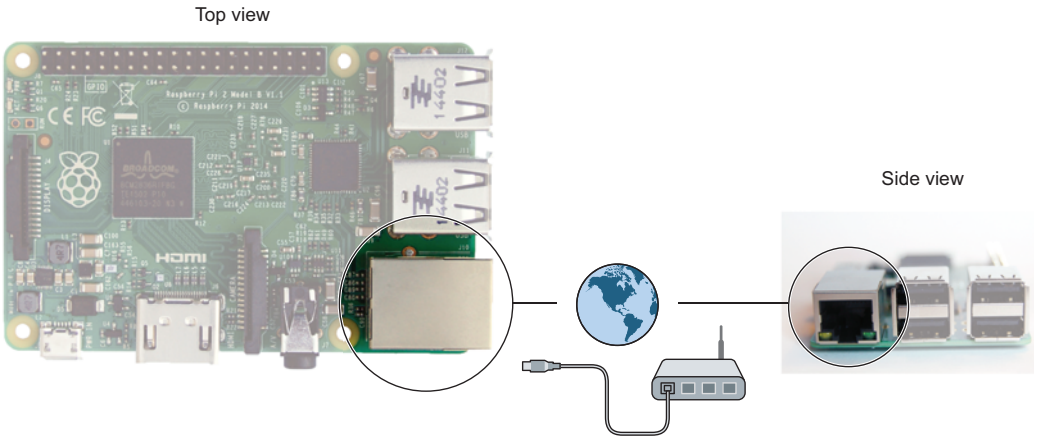


Figure B.7 The Ethernet port on the Raspberry Pi supports connecting a Pi to a home network. Connect an Ethernet cable from your Pi to your router or modem to access the internet. With your Pi connected to a network, you can remotely connect to the Pi from another computer using special programs such as SSH and VNC Server.

Ethernet port

Having a connection to the internet lets you use your Pi to surf the web and download software; you can even control your Pi from another computer. Your Raspberry Pi's Ethernet port is located next to the USB ports (see figure B.7). Using the Ethernet port is an easy way to connect a Raspberry Pi to the internet. The only trouble is that you'll need to have your Pi where an Ethernet cable connection can reach it.

TV or monitor connection options

It's easiest to connect your Pi to a TV or monitor if it has an HDMI port or DVI port; this is covered in chapter 1. But what if you don't have one of those ports? There are other ways to connect your Pi. Let's first identify a couple different types of ports you might see on the back of your TV or monitor and then learn how to connect your Pi to them.

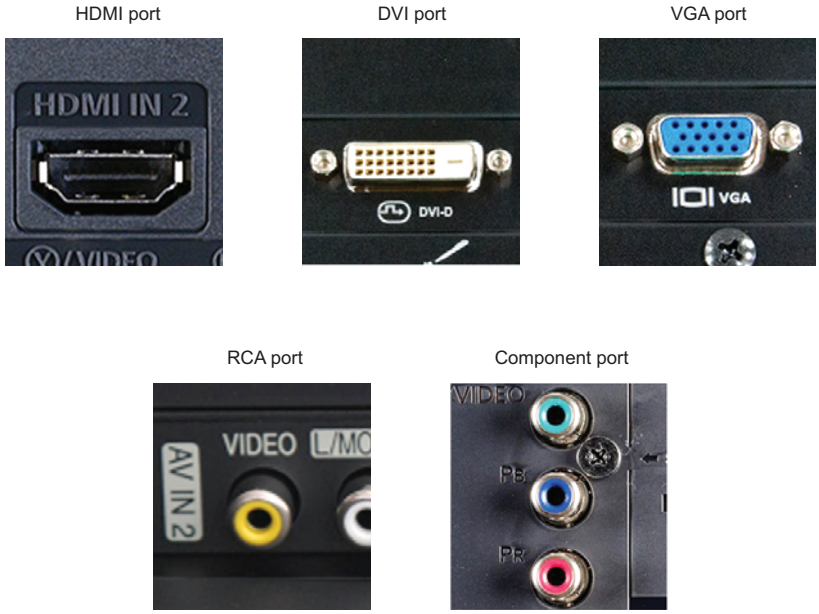


Figure B.8 Common types of video input ports found on TVs and monitors. A Raspberry Pi can connect to any one of these ports. Some ports (DVI, VGA, RCA [or composite], and component) require using special adapters or converters with a Pi.

IDENTIFYING PORTS AND MAKING THE CONNECTION

Take time to study the connections on your TV or monitor. Try to identify the video ports, comparing them to the pictures of connectors in figure B.8.

For certain ports, you may need to buy an adapter that converts one type of port to another. We'll cover VGA, RCA, and component ports. See chapter 1 for the HDMI and DVI port connections.

RCA PORT

This type of port is a yellow, round connector. It's usually found next to red-and-white RCA audio connectors.

You'll need to purchase a special cable that is a 3.5 mm four-pole plug at one end and an RCA composite video and audio cable at the other

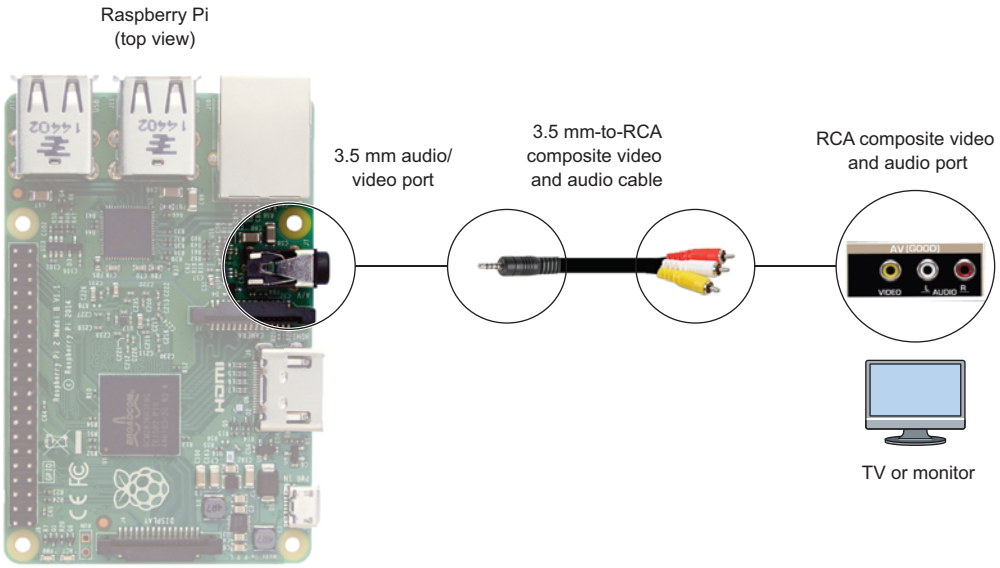


Figure B.9 The Raspberry Pi can be connected to a TV or monitor using an RCA video cable. The cable connects from the Pi's 3.5 mm audio/video port to the RCA video input port on the TV or monitor. Using the RCA connection produces a low-quality picture but can be a good option if you don't have a TV or monitor that supports HDMI.

end. Plug the cable into the 3.5 mm audio/video port, and plug the other end into your screen's composite video input. Typically, the screen will have red-and-white audio-input connectors next to the video input. Connect the red-and-white RCA audio connectors if you want to have sound as well (see figure B.9).

VGA PORT

A VGA port has a flat top and bottom with sides that slant inward. The port has three rows of five round pin holes. Connecting a Pi to a TV or monitor with a VGA port isn't recommended because you'll need to purchase an adapter and may run into potential issues with configuring your Pi to detect your monitor. If you decide to try this option, you'll need an HDMI-to-VGA adapter. You'll also need to update the configuration settings on your Raspberry Pi. This isn't covered in this

book, but the Raspberry Pi forums can provide you with more information on altering the configuration settings to use an HDMI-to-VGA adapter.

COMPONENT VIDEO INPUT

A component video port on a TV has a set of three round connectors that are green, blue, and red. Using this port isn't recommended because of the additional cost of a converter and because you may have to do additional configuration of your Pi to successfully connect to your monitor. If you decide to use this option, you'll need a component-to-HDMI converter. Such a converter should come with its own power supply. Avoid ones that don't, because they won't work with your Pi. The converter will cost you around $50, so if you have other options, save your money—try using a different TV or monitor, or put that money toward a new or used LCD or LED monitor for your Pi.

With the ports covered, let's examine the differences between the Raspberry Pi 2 Model B and older model boards.

Legacy boards

The Raspberry Pi is made by the Raspberry Pi Foundation, and several versions and models have been released over the last several years. We'll show and discuss the major differences between the boards.

Raspberry Pi 1 Model B

The Raspberry Pi 1 Model B was the version of the Pi that many came to love. The Pi was originally conceived to help develop a new generation of programmers and hackers, but it was unexpectedly popular with many hobbyists and entrepreneurs because of all the great things they could do and make with it. The board looks a bit different from the Raspberry Pi 2 Model B (see figure B.10)

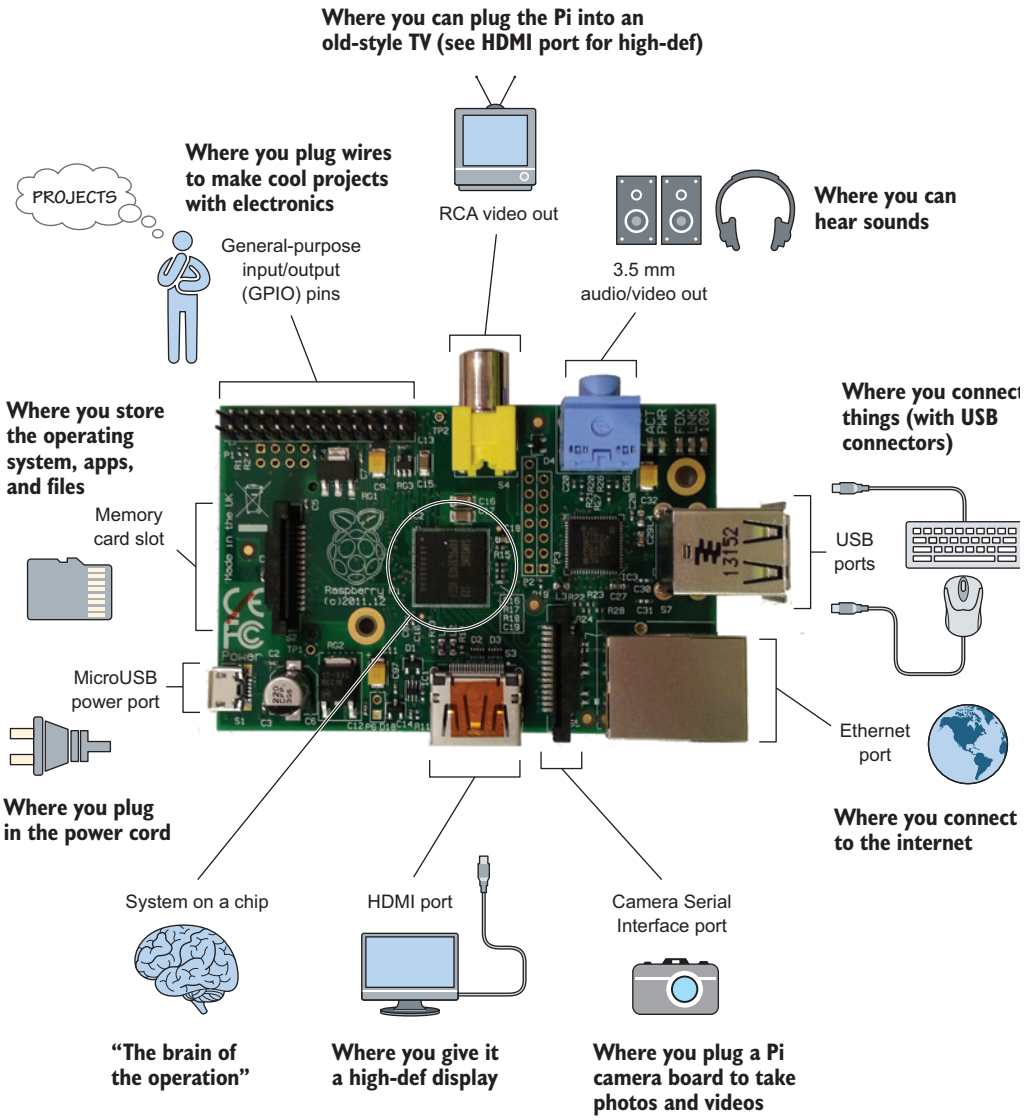


Figure B.10 The Raspberry Pi 1 Model B has been wildly popular. It has been used for a spectrum of applications from scientific research to art and education.

Here are a few key differences between the Raspberry Pi 1 Model B and the Raspberry Pi 2 Model B:

- *USB ports*—The Pi 1 Model B has only two USB ports. This makes it challenging to connect a keyboard, a mouse, and a USB Wi-Fi adapter. A great workaround is to use a powered USB hub to connect more USB devices.
- *RCA (or composite) video out*—The Pi 1 Model B has a dedicated RCA connector to connect it to old-style TVs. The Pi 2 Model B has integrated this into the 3.5 mm audio/video port.
- *System on a chip*—The earlier Pi Model B uses a single-core 700 MHz processor, whereas the Pi 2 Model B uses a quad-core 900 MHz processor. Thus the newer model is about four times faster.
- *Memory card slot*—The Pi 1 Model B used a standard size SD card. The Pi 2 Model B uses a mini-SD card slot that has a spring mechanism to hold the card in securely.
- *GPIO pins*—The number of pins and how they're numbered is different on the earlier Model B. There are only 20 pins on the older model; the newer model has 40 pins. If you're working with a Pi 1 Model B, refer to online references for the pin numbering.

Raspberry Pi 1 Model B+

After the Raspberry Pi 1 Model B came the Raspberry Pi 1 Model B+. The boards look very different. In contrast, if you compare the Raspberry Pi 1 Model B+ to the later Raspberry Pi 2 Model B, they're nearly identical—in terms of available ports and the location of those ports, they're exactly the same. Figure B.11 shows the Raspberry Pi 1 Model B+.

The key differences from the Raspberry Pi 2 Model B are as follows:

- *System on a chip*—The B+ has a single-core, 700 MHz processor, whereas the Pi 2 Model B has a quad-core, 900 MHz processor.
- *Working memory (RAM)*—The Model B+ has 512 MB compared to the Pi 2 Model B's 1 GB.

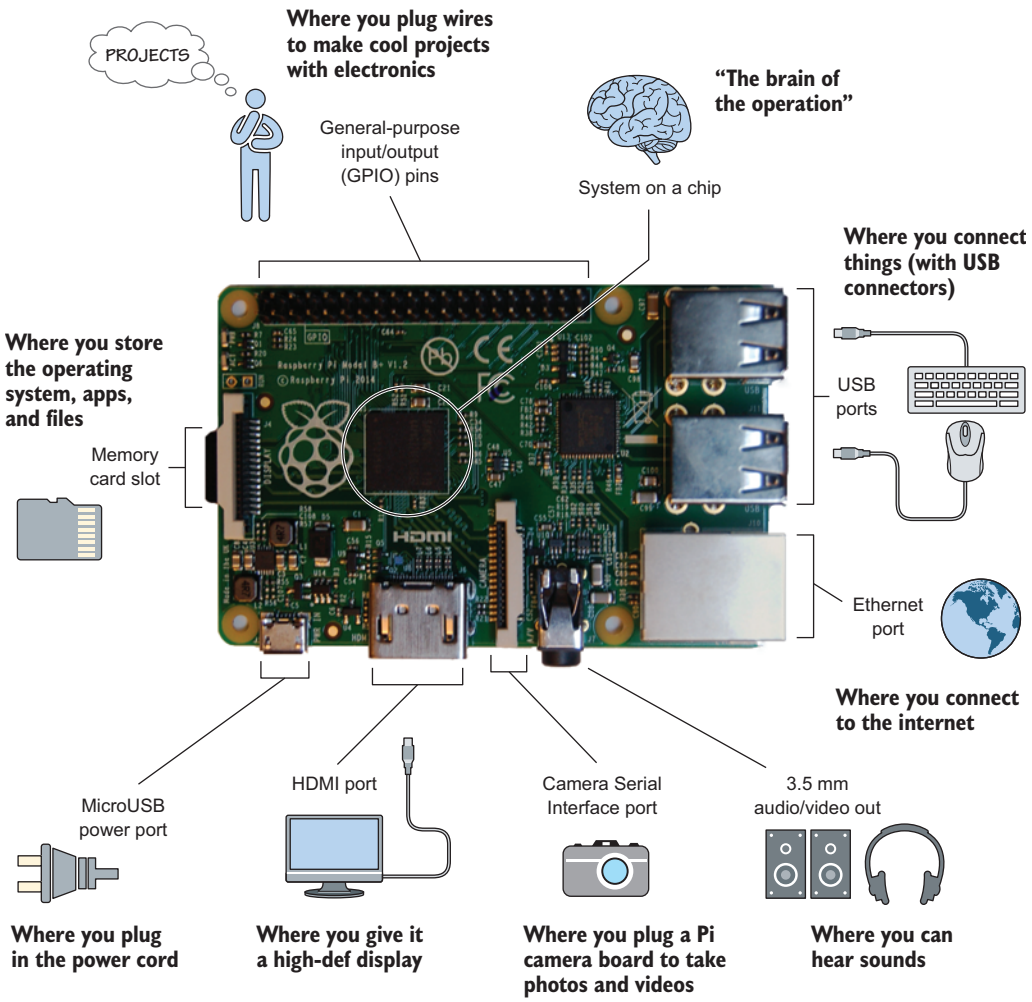


Figure B.11 The Raspberry Pi 1 Model B+ was a major revision of the Raspberry Pi 1 Model B. It increased the number of USB ports from two to four, added more pins for GPIO, and changed to a microSD memory card slot. The ports on the Raspberry Pi 1 Model B+ are the same as those on the Raspberry Pi 2 Model B.

Other boards

We aren't covering the Raspberry Pi Model A or A+, but many of the ports are the same. The main difference is that the Model A and A+ have only one USB port, no Ethernet port, and less working memory (RAM)—256 MB. The Model A and A+ are useful when you have a project that needs a smaller computer that requires less power than the Model B or B+.

Appendix C

Solutions to chapter challenges

In this appendix, you'll find answers to the challenges presented at the end of each chapter. For challenges that require more lines of code than will fit on a page, I provide hints and snippets of code. The complete programs for the solutions are found in the code download that goes with this book. Comments are included in the code to help you understand the design and function of the programs. The solutions to the challenges are organized by chapter. Let's begin!

Chapter 1

At the end of the first chapter, you go on a scavenger hunt:

- *Squirrel*—To find the squirrel game, choose Menu > Games > Python Games. After you select how you would like sound (audio) to be output, you'll see a list of Python games. The squirrel game is near the middle of the list. Win the game, and achieve Omega Squirrel.
- *Calculator*—Select Menu > Accessories > Calculator. 89 x 34 is 3,026.
- *Shutdown*—Shut down or restart your Raspberry Pi by choosing Menu > Shutdown. The shutdown menu lets you choose to shut down, reboot, or log out.

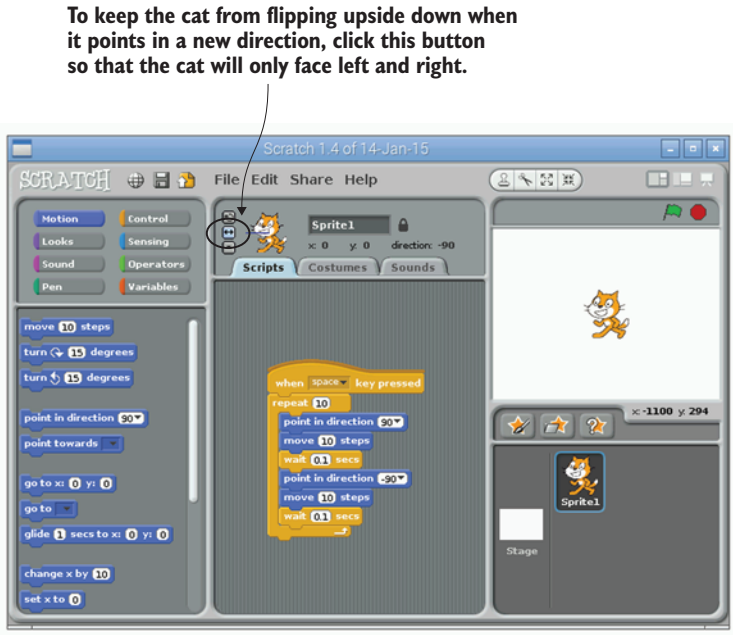


Figure C.1 Make the cat dance in Scratch by dragging program blocks into the script area.

- *Black desktop*—To change the desktop background to black, right-click anywhere on the desktop and select Desktop Preferences. In the Desktop Preferences window, look in the middle of the screen for a Background Color label. Click the white box to select a new background color. Click OK to select the color, and then click Close to close the Desktop Preferences window.
- *Scratch bonus*—To open Scratch, select Menu > Programming > Scratch. When Scratch opens, construct a program by dragging blocks into the script area for your cat sprite. Figure C.1 shows an example of a dancing cat program that makes the cat dance back and forth 10 times when the space bar is pressed.

Chapter 2

The challenges in this chapter are about displaying characters to the screen and doing some mathematics.

The Matrix

Create a screen full of 1s and 0s by using the `print` function and the multiplication operator like this:

```
matrix = "0100101101001100100110001011001011110000010101"
print(matrix * 100)
```

Building a brick wall

To solve this one, create a variable named `brick` and give it a string of characters like this:

```
brick = "|__"
print(brick * 1000)
```

To make the bricks look like raspberries, you could try

```
brick = "|_o88{_"
print(brick*300)
```

Use your imagination to visualize that this is a sideways raspberry brick. The bracket is the leaf on top of the raspberry.

Pi electrons

You're trying to figure out how many electrons per second it takes to equal 1 amp flowing into your Raspberry Pi. The calculation using Python looks like this:

```
>>> electron_charge = 1.60 * 10**-19
>>> electrons_flowing = 1 / electron_charge
>>> print(electrons_flowing)
6.249999999999999e+18
```

The answer is 6,250,000,000,000,000,000. That's a lot of electrons!

Chapter 3

These challenges are about gathering input, joining together strings, and displaying text to the screen.

Knight's Tale Creator 3000

To make this program, you want to first print a title and then gather a series of words from the player:

```
title = "Knight's Tale Creator 3000"
print("*" * 80)
```

```
print(title)
print("*" * 80)

player_name = input("Enter your name: ")
adjective = input("Enter an adjective: ")
famous_person = input("Enter the name of a famous person: ")
animal = input("Enter the name of an animal: ")
vacation_place = input("Enter a place you would go on vacation: ")
sharp_thing = input("Enter the name of something sharp: ")
exclamation = input("Enter something you might exclaim aloud: ")
```

Gather input from the player and store it in variables.

Next, you join the input words with the story. You can do this sentence by sentence to make the code a bit easier to follow:

```
sentence1 = "There was a brave knight, " + player_name + ", who was
➥ sent on a quest to vanquish the " + adjective + " evildoer,
➥ " + famous_person + ". "
sentence2 = "Riding on his/her trusty " + animal + ", the brave " +
➥ player_name + " traveled to the faraway land of " + vacation_place
➥ + ". "
sentence3 = player_name + " battled valiantly against " + famous_person +
➥ "'s army using his " + sharp_thing + " until he defeated them. "
sentence4 = "Emerging victorious, " + player_name + " exclaimed, '" +
➥ exclamation + "!!!' I claim the land of " + vacation_place + " in the
➥ name of Python."
```

Finally, let's join the sentences and display the tale to the screen:

```
tale = sentence1 + sentence2 + sentence3 + sentence4
print(tale)
```

Subliminal messages

You're trying to create a message that's hidden in a large display of characters. You start by asking for the person's name and something they would like:

```
title = "Subliminal Messages"
print("*" * 80)
print(title)
print("*" * 80)

player_name = input("Enter your name: ")
thing = input("Enter the name of something you want: ")
```

Gather input from the player.

Next, create a pattern of letters, numbers, and symbols in which you'll hide the message:

```
weird_characters = "*#ad@32*)23 )@*sad# 2&^ 32^423!"
```

Finally, you create the full message by making it read "You really want to buy [player_name] a [thing]", but hide it by printing out on the screen the pattern of characters before and after the message:

Create the message.

```
message = "You really want to buy " + player_name + " a " + thing + "."
print(weird_characters * 10 + message + weird_characters * 10)
```

Hide it between the weird characters.

Chapter 4

This chapter's challenge is about using some of your new skills, like `if`/`then` (or conditional) statements, as well as toolboxes like the `random` module.

Rock, Paper, Scissors!

For this challenge, you start by importing the `random` module, creating a title, and defining any variables you're going to need:

```
import random

play_again = "Yes"
choices = ["Rock","Paper","Scissors"]
```

Next, you want to display the title and then start a `while` loop that will gather the player's choice and get a random computer choice:

```
title = "Rock, Paper, Scissors!"
print("*" * 80)
print(title)
print("*" * 80)

while play_again == "Yes":
    print("Choose Rock, Paper, or Scissors:")
    player_choice = input("Enter your choice: ")
    computer_choice = choices[random.randint(0,2)]
```

Get a random computer choice.

Get the player's choice.

You then want to display the two choices and use an `if` statement to test whether the player and computer choices are the same. If not, you

want to check whether you have one of the following Player versus Computer combinations:

- Rock (Player) beats scissors (Computer)
- Scissors (Player) beats paper (Computer)
- Paper (Player) beats rock (Computer)

You program all this inside the `while` loop because you want it to be repeated as long as the player wants to play. At the end of the loop, the player is asked if they want to play again:

Display the two choices.

```
    print("You choice is " + player_choice + ".")
    print("The computer's choice is " + computer_choice + ".")
    if player_choice == computer_choice:
        print("It's a tie")
    else:
      if ((player_choice == "Rock" and computer_choice == "Scissors") or
          (player_choice == "Scissors" and computer_choice == "Paper") or
          (player_choice == "Paper" and computer_choice == "Rock")):
            print("!" * 80)
            print("You win!")
            print("!" * 80)
        else:
            print(":(" * 40)
            print("You lose!")
            print(":(" * 40)
play_again = input("Do you want to play again [Yes/No]? ")
```

You create a large `if` statement that tests whether `player_choice` and `computer_choice` form one of the winning combinations. Each of the combinations is wrapped in parentheses, and you use `or` between them. This ensures that if any one of the combinations is correct, the winning message will be displayed.

Chapter 5

Introducing dramatic pauses

You were given some good hints about how to do this in chapter 5. Rather than display too much code here, I suggest that you head over to the code download to see what this looks like.

Random demise

In this challenge, you're creating a more random and exciting end for your adventurer, Raspi. For this, you need to import the `random` module at the top of your program, define some new variables for the different endings, and create a list of the endings. This will allow you to have the computer pick a number:

```
import time
import random

demise1 = """Raspi sees a rock on the ground and picks it up. He feels a
➥ sharp pinch and drops the rock. Just then he realizes it wasn't a rock
➥ but a poisonous spider as he collapses to the ground."""
demise2 = """Standing in the cave, Raspi sees a small rabbit approach
➥ him. Raspi gets a bad feeling about this rabbit. Suddenly, the
➥ rabbit attacks him, biting his neck."""
demise3 = """Whoa, is that a piece of gold? As Raspi walks over to it,
➥ he doesn't see a hole in the floor. Suddenly, he falls down the hole,
➥ never to be heard from again."""
endings = [demise1, demise2, demise3]
```

You use triple quotation marks, called *string literals*, to make strings that span multiple lines. You also store the three different endings in a list called `endings`.

Finally, to solve this challenge, change the `wrong_answer` function to get a random number, select an ending, and then display it to the screen:

```
def wrong_answer():
    print("You seem to have trouble making good decisions!")
    time.sleep(2)
    random_ending = endings[random.randint(0,2)]
    print(random_ending)
    time.sleep(2)
    print("Game Over!!!")
```

Select a random ending from the list of endings.

Check the code download to see how it all works together.

Play again?

To add a play-again option, add a new variable at the top of your program and set it equal to `"Y"` to start:

```
play_again = "Y"
```

Next, put all of your cave-selection logic in a `while` loop. The `while` loop will depend on the value of `play_again`, but you'll use the string function `upper()` to make the `play_again` value all uppercase. This helps if the user accidentally enters `y` instead of `Y`:

```
while play_again.upper() == "Y":
```

At the end of the `while` loop, you also need to ask the user if they want to play again. Store their response in the `play_again` variable:

```
print("Do you want to play again?")
play_again = input("Enter Y for yes or N for no: ")
```

Another part of this challenge is adding a scream sound (or any other sound you want). First, make sure you import the `os` module and set up a variable with a sound file:

```
import os
scream_file_path =
➥ "/usr/share/scratch/Media/Sounds/Human/Scream-male2.mp3"
```

Add a new line to the `wrong_answer` function that calls OMXPlayer and tells it to play the scream sound:

```
os.system("omxplayer " + scream_file_path)
```

Run the program to test it out! Add it to all the other game-over endings in the game to make it even more fun!

> NOTE Make sure you have speakers or headphones connected, or you won't hear anything.

Chapter 6

Wave pattern

Let's turn on each LED one by one. Then, when they're all on, you'll turn them off one by one. Each light is turned on or off by setting its state to `HIGH` (on) or `LOW` (off). You create the sequence by adding a time delay between each command:

```
while True:
    GPIO.output(LED_pin_red, GPIO.HIGH)
    time.sleep(1)
```

Loop to blink the LEDs in a wave pattern.

Turn on the LEDs one by one.

```
GPIO.output(LED_pin_green, GPIO.HIGH)
time.sleep(1)
GPIO.output(LED_pin_blue, GPIO.HIGH)
time.sleep(1)
GPIO.output(LED_pin_red, GPIO.LOW)
time.sleep(1)
GPIO.output(LED_pin_green, GPIO.LOW)
time.sleep(1)
GPIO.output(LED_pin_blue, GPIO.LOW)
time.sleep(1)
```

Turn off the LEDs one by one.

You can adjust the sleep time to get a faster or slower animation.

Simon says

In this challenge, you're creating a program that blinks lights in a pattern like the classic game *Simon*. As before, the program requires that the GPIO and time modules be imported and the GPIO pins be set up properly. See the code download for the full code listing. Start by defining the `simon_says` function:

```
def simon_says(color1, color2, color3, color4, color5):
    colors = [color1, color2, color3, color4, color5]
    for i in range(0,5):
        color = colors[i]
        if color == "red":
            GPIO.output(LED_pin_red, GPIO.HIGH)
            time.sleep(1)
            GPIO.output(LED_pin_red, GPIO.LOW)
        elif color == "green":
            GPIO.output(LED_pin_green, GPIO.HIGH)
            time.sleep(1)
            GPIO.output(LED_pin_green, GPIO.LOW)
        elif color == "blue":
            GPIO.output(LED_pin_blue, GPIO.HIGH)
            time.sleep(1)
            GPIO.output(LED_pin_blue, GPIO.LOW)
        time.sleep(1)
```

Create a list with the five colors.

Loop through the five colors.

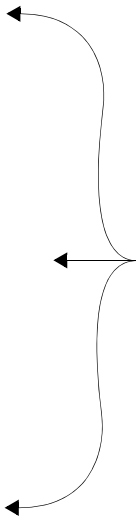
Grab the name of a single color and turn the color on and off.

This creates a list with all the colors and loops through them one by one. For each one, the function checks its value and turns on and off the LED of that color. To use the function, you call it and give it the pattern you want to create, along with some helpful messages:

```
print("Ready for #1!")
time.sleep(1)
print("Simon Says: red, green, red, red, blue")
time.sleep(1)
print("Watch my lights!")
time.sleep(1)
simon_says("red", "green", "red", "red", "blue")

print("Ready for #2!")
time.sleep(1)
print("Simon Says: blue, green, blue, green, red")
time.sleep(1)
print("Watch my lights!")
time.sleep(1)
simon_says("blue", "green", "blue", "green", "red")

print("Ready for #3!")
time.sleep(1)
print("Simon Says: green, blue, blue, red, green")
time.sleep(1)
print("Watch my lights!")
time.sleep(1)
simon_says("green", "blue", "blue", "red", "green")
time.sleep(1)
print("Thank you for playing!!!")
```



Go, Simon, go!

Random blinking

This challenge is about blinking LEDs on and off for random amounts of time between 0 and 3 seconds. Let's see how to do it. I won't show the top part of the program with the typical setup of the GPIO pins; refer to the code download for the full code listing. At the top of your program, don't forget to import the `random` module so you can use it to generate random numbers:

```
import random
```

To accomplish this challenge, you need to create two variables and store in them a random number between 0 and 3. These variables are the amount of time the lights should stay on and off:

```
on_random_time = random.random() * 3
off_random_time = random.random() * 3
```

Next, you can use the random time with the `sleep` function to make the light blink. Put this inside a loop, making sure that each time through the loop, new random on and off times are created:

```
while True:
    on_random_time = random.random() * 3
    off_random_time = random.random() * 3

    GPIO.output(LED_pin_red, GPIO.HIGH)
    GPIO.output(LED_pin_green, GPIO.HIGH)
    GPIO.output(LED_pin_blue, GPIO.HIGH)
    time.sleep(on_random_time)

    GPIO.output(LED_pin_red, GPIO.LOW)
    GPIO.output(LED_pin_green, GPIO.LOW)
    GPIO.output(LED_pin_blue, GPIO.LOW)
    time.sleep(off_random_time)
```

Loop to blink the LED.

Get a random number.

Turn the lights on for a random amount of time.

Turn the lights off for a random amount of time.

The off and on times change each time through the loop. Enjoy some fun blinking!

Chapter 7

The chapter challenges involve using your Guessing Game and controlling the RGB LED.

Game winner

Let's write a function to quickly flash the RGB LED three different colors. Define a new function called `winning_flash`:

```
def winning_flash():
    for i in range(0,20):
        GPIO.output(LED_pin_red, GPIO.HIGH)
        time.sleep(0.05)
        GPIO.output(LED_pin_red, GPIO.LOW)
        time.sleep(0.05)

        GPIO.output(LED_pin_green, GPIO.HIGH)
        time.sleep(0.05)
        GPIO.output(LED_pin_green, GPIO.LOW)
        time.sleep(0.05)
```

Function to create a winning flash sequence.

Flash three different colors 20 times.

Flash the red LED.

Flash the green LED.

```
        GPIO.output(LED_pin_blue, GPIO.HIGH)
        time.sleep(0.05)
        GPIO.output(LED_pin_blue, GPIO.LOW)
        time.sleep(0.05)
```

Flash the blue LED.

If you need help figuring out where to add this function and call it in your code, check the code download for more answers. You add it to the `if` statement when `guess` is equal to `number_in_my_head` so you get a wonderful flashing celebration when you win.

Easter egg

To make an Easter egg in your program, you need to have the code check to see whether the player entered a certain value instead of the usual number guess. Edit the main portion of the logic for the LED Guessing Game program to first check whether the secret word was entered. If it wasn't, the program continues to convert the input text into an integer and check whether the guess was correct, too high, or too low. If the player enters the word *Spam*, you call an `easter_egg` function:

```
while count_guesses < 6:
        guess = input("guess " + str(count_guesses) + " - What is your
        ➥ guess?: ")
        if guess == "Spam":
            easter_egg()
        else:
            guess = int(guess)
            count_guesses += 1
            if guess == number_in_my_head:
                flash(LED_pin_green)
                print("You won!  No doom for you!")
                break
            elif guess > number_in_my_head:
                flash(LED_pin_red)
            elif guess < number_in_my_head:
                flash(LED_pin_blue)
    else:
        game_over()
```

Check to see if "Spam" was entered.

Call the easter_egg function.

As a special bonus, you can create an `easter_egg` function that displays a Spam song or whatever message you'd like.

```
def easter_egg():
    crazy_flash()
    print("Easter Egg!!!")
    time.sleep(1)
    print("""
            Spam spam spam spam.
            Lovely spam!
            Wonderful spam!
            Spam spa-a-a-a-a-am spam spa-a-a-a-a-am spam.
            Lovely spam!
            Lovely spam!
            Lovely spam!
            Lovely spam!
            Lovely spam!
            Spam spam spam spam!
            """)
```

Define a special Spam Easter egg.

In the `easter_egg` function, you call a `crazy_flash` function. The one shown here makes the RGB LED quickly flash purple and green. It's similar to how you created the `winning_flash` function:

Function to create a crazy flash sequence.

```
def crazy_flash():
    for i in range(0,20):
        GPIO.output(LED_pin_red, GPIO.HIGH)
        GPIO.output(LED_pin_blue, GPIO.HIGH)
        time.sleep(0.05)
        GPIO.output(LED_pin_red, GPIO.LOW)
        GPIO.output(LED_pin_blue, GPIO.LOW)
        time.sleep(0.05)

        GPIO.output(LED_pin_green, GPIO.HIGH)
        time.sleep(0.05)
        GPIO.output(LED_pin_green, GPIO.LOW)
        time.sleep(0.05)
```

Flash different colors 20 times.

Flash the red and blue LED together.

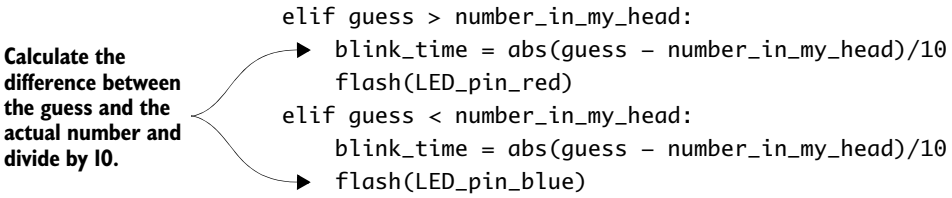
Flash the green LED.

Create your own `easter_egg` and `crazy_flash` functions, or see the code download for example ones that you can modify.

Warmer and colder

Let's alter the guessing game to flash slower if you're colder or further from the correct answer, and flash faster if you're warmer or closer to

the correct answer. Add some calculations so that `blink_time` is determined by the difference between the guess and the correct answer:




```
elif guess > number_in_my_head:
    blink_time = abs(guess - number_in_my_head)/10
    flash(LED_pin_red)
elif guess < number_in_my_head:
    blink_time = abs(guess - number_in_my_head)/10
    flash(LED_pin_blue)
```

The `abs` function gets the absolute value—the distance a number is from zero. You need to do this because you can't tell your Pi to sleep for a negative amount of time. That would be silly! You make this addition for both cases: when the player's guess is higher and lower than the actual number. You divide the numbers by 10 to speed up `blink_time` and make sure your light isn't blinking too slowly.

Finally, a nice touch is to add extra information to the game instructions so the player knows the blinking speed gives them a hint about how close or far they are. See the code download for an example.

Darth Vader surprise

Using what you learned in chapter 7 and a couple of new things, let's see if you can make a Darth Vader image pop up on the screen when you lose the game. You'll need an internet connection for the next few steps. Download a good Darth Vader image from the web, and make sure to save it to the home\pi folder where your Python programs are located. Take special note of the filename.

After downloading the image, install the `fim` image-viewing software on your Raspberry Pi:

```
pi@raspberrypi ~ $ sudo apt-get -y install fim
```

> NOTE Make sure you include a line at the top of the program to import the `os` module.

When it's done, test that `fim` works from Terminal:

```
pi@raspberrypi ~ $ fim Darth_Vader.jpg
```

NOTE When `fim` is running, you need to press Esc (escape) to exit.

When you exit `fim`, the screen will display remnants of the image. It's a funny issue, which you can fix by grabbing one of your windows by the title bar and swiping it around the screen to erase the image remnants and return it to the normal Raspbian desktop appearance.

In the Guessing Game, because you need to call `fim` from your Python program, add a line to import the `os` module at the top of the program:

```
import os
```

Next, edit the `game_over` function to display the image. The `game_over` function is called only when the player guesses incorrectly five times:

```
def game_over():
    print("You lost!")
    print("Better luck next time!")
    time.sleep(2)
    os.system("fim -a Darth_Vader.jpg")
```

Notice that you use `fim` with the `-a` option to display Darth Vader. This option automatically scales the image to fill the full screen. Here are some commands you can use to rotate or resize the image when it's displayed on the screen:

Option	Result
`+/-`	Zoom in/out
`A`	Automatically scale
`F`	Flip
`M`	Mirror
`R/r`	Rotate 10 degrees clockwise / counterclockwise
`Esc/q`	Quit

Test it to see if it works!

Chapter 8

Let's see what fun things you can do with buttons.

Double button press surprise

This challenge involves taking the project from the chapter and making something new and different happen when both buttons are pressed at the same time. In this case, you'll make your Pi play a percussion sound to go with your vocals and music. You don't need to change any of the wiring because you already have the two buttons.

First let's add some code at the top of the program to create a path to where the sound effects are stored:

```
path_effects = "/usr/share/scratch/Media/Sounds/Effects/"
```

Next get a list of effects from the folder and store the list in a variable, `sounds_effects`. Put this next to where you load the other lists:

```
sounds_effects = get_MP3_sounds(path_effects)
```

Finally, you need to tell your program to check whether button 1 and button 2 are pressed. You're going to modify the main game loop to first check if both are being pressed. You use the `if`/`elif` statement for this. Use the Boolean "and" operator—the ampersand (`&`)—to make this `if` statement true only if both button 1 and button 2 are pressed. If they aren't, the statement will next check button 1, and finally it will check button 2:

Start an infinite loop (must use Ctrl-C to stop it).

```
while True:
    if GPIO.input(button_pin1) & GPIO.input(button_pin2):
        #print("You pressed both #1 and #2!")
        play_random_sound(path_effects, sounds_effects)
        time.sleep(.1)
    elif GPIO.input(button_pin1):
        #print("You pressed #1!")
        play_random_sound(path_music, sounds_music)
        time.sleep(.1)
    elif GPIO.input(button_pin2):
        #print("You pressed #2!")
        play_random_sound(path_vocals, sounds_music)
        time.sleep(.1)
    time.sleep(.1)
```

Pause slightly before checking the button for input again.

Let's test to see if it works! Check out the code download if you need further details on the program.

Yoda Magic 8 Ball

Before you dive into the programming for this challenge, you need to work on the hardware. Because this challenge needs only one button, remove button 2 from the breadboard, along with its jumper wires and resistor. Your breadboard should have one button now, connected to GPIO 6.

Next, gather a set of Yoda sounds. You can download sounds from Soundboard once you create a free account. For this example solution, you'll use five sound files, but feel free to use any ones you want. Make sure they're MP3 sound files so they'll work with OMXPlayer. The Yoda sound files in this example solution are as follows:

- Fear in You.mp3
- I am strong.mp3
- No.mp3
- Patience.mp3
- Use the Force.mp3

Much like the classic Magic 8 Ball game, the answers are sometimes clear and other times strange or unclear.

As in the DJ Raspi project, you need to import several modules for this project, set up your Pi's GPIO pin for input (detecting electrical signals), and create some variables. Most notably, you need to create a variable for the folder with your Yoda sound files:

```
import RPi.GPIO as GPIO
import time
import random
import os

button_pin = 6
play_again = "Y"

GPIO.setmode(GPIO.BCM)
GPIO.setup(button_pin,GPIO.IN)

path_yoda = "/home/pi/yoda/"
```

Import the libraries you need.

Variable for the GPIO pin used by the button.

Set the path to the Yoda sound files.

Use the same `get_MP3_sounds` and `play_random_sound` functions from your DJ Raspi project. One slight improvement you can make to the `play_random_sound` function is to hide the messages that OMXPlayer displays on the screen (they make it harder to read what the game is telling you to do). Change this one line to divert all the output messages to an empty or null location:

```
def play_random_sound(sound_path, sound_files):
    random_sound_index = random.randint(0,len(sound_files)-1)
    # print("Playing: " + sound_files[random_sound_index])
    os.system("omxplayer -o local '" + sound_path +
              "/" + sound_files[random_sound_index] + "' >/dev/null")
```

This is a great example of being able to reuse code! Next, you'll gather the list of MP3 Yoda sounds from the folder.

```
sounds_yoda = get_MP3_sounds(path_yoda)
```

After printing out a nice title, you then show instructions to the player and enter the main loop that checks whether the button was pressed. In this loop, you call the `play_random_sound` function so the Raspberry Pi responds with an answer to the player's question:

```
print("*" * 80)
print("Ask aloud a Yes or No question, then press the button: ")
print("*" * 80)

while play_again.upper() == "Y":
    if GPIO.input(button_pin):
        print("Yoda is considering your question...")
        time.sleep(1)
        print("Listen to Yoda's answer:")
        time.sleep(.5)
        play_random_sound(path_yoda, sounds_yoda)
        print("*" * 80)
       print("Ask aloud a Yes or No question, then press the button: ")
        print("*" * 80)
else:
    print("Thank you for consulting Yoda!")
```

Check if the button has been pressed.

Enjoy making your future decisions with the help of Yoda!

Appendix D

Raspberry Pi projects

In this appendix, you'll find short discussions and descriptions of projects you can do with your Raspberry Pi. The goal is to launch you on your way. This isn't a detailed set of instructions, but rather hints and basic steps for how you can make some of these projects.

Halloween heads

Halloween can be an inspiring time to use your Raspberry Pi to create a fun or scary display for your home. Let's face it—it's fun to scare people on Halloween. This project is about building a system for surprising trick-or-treaters who come to your door. When they approach, their movement will trigger a motion sensor that will display a video of a face talking or singing. The video is projected onto a Styrofoam head that is placed next to the door.

Here is what you'll need for this project:

- Raspberry Pi with a breadboard, a breakout board, and a ribbon cable
- Passive infrared (PIR) motion sensor
- Projector
- Powered computer speakers
- Styrofoam heads (one or more)

- Small tables: one for the Pi and projector, and another for the Styrofoam head
- Extension cord and power strip
- Video of a singing or talking head

To construct this project, here are the steps:

1. Connect your Raspberry Pi to the breadboard, and add the PIR sensor. This is similar to how you added the mini pushbutton in chapter 8.
2. Download a video with a talking or singing head, or record your own. Write a Python program to play the video when the PIR sensor is triggered. This is similar to the DJ Raspi program, which plays a sound when the button is pressed.
3. Test your program with the sensor and video working together.
4. Set up a small table about 10 feet from your front door. On the table, set up your Raspberry Pi, breadboard with PIR sensor, speakers, and projector. Place the PIR sensor so that it will detect motion as someone approaches the door. Use an extension cord to provide the electrical power needed. (Only set this up if no rain is predicted!)
5. Set up another small table or box next to your door. Place the Styrofoam head on it. Position the head so that the projector's video displays the face on the head. Test and adjust the projector and the positioning of the head so that everything is aligned. When the video plays, the head will appear to come alive!

Here are a couple of key resources that may help you with this project:

- Visit the SparkFun website at www.sparkfun.com and search for PIR sensors. This company has lots of great components that can help you make almost any electronics project you can imagine.
- You can make the whole screen blank (all black) by using OMXPlayer with the blank option like so:

```
pi@raspberrypi ~ $ omxplayer -b singheads.mp4
```

Time-lapse photography

You can easily connect a high-definition camera to your Raspberry Pi that is capable of taking digital photographs or videos (see appendix B for more information). In this project, you explore how you can set up your Pi to take time-lapse photographs.

Time-lapse photography typically involves taking a series of photographs and then stitching them together into a video. The individual photographs may be taken seconds, minutes, hours, or days apart. This technique is commonly used to show an accelerated view of something happening. Here are some examples of time-lapse scenes:

- A glacier slowly retreating over the course of a year
- The sun rising and setting, and the moon rising and setting
- A plant growing

Here are some simple steps to get started with a time-lapse photography project:

1. Set up your Raspberry Pi with the Pi camera kit, and test that it's working.
2. The subject of your time lapse determines how you need to mount the Pi camera. The camera doesn't come with a case or any way to hold it up, so you'll need to engineer a mount of some kind. Cardboard, hot glue, craft sticks, and duct tape are all great materials for fabricating something to hold up the camera. LEGO blocks can also be a useful material.

   If you're going to leave the camera outside for a long time, consider whether you'll need to waterproof your Raspberry Pi. Plastic containers left over from takeout food can make a great case; you'll just need to make holes in the container for wires and seal any gaps with hot glue.
3. Plan how to get electrical power to your Pi. That may determine where you set up the Pi and camera.

4 Program your Raspberry Pi to take the photographs and store them in a folder. Open LXTerminal, and install the `picamera` module for Python 3.X:

```
pi@raspberrypi ~ $ sudo apt-get install python3-picamera
```

To get you started, you can use a program like this to capture a series of photographs. This example takes a photograph every 3 minutes:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.start_preview()
    time.sleep(2)
        for filename in camera.capture_continuous
        ➥ ('image{counter:04d}.jpg'):
        print('Captured %s' % filename)
                time.sleep(180)
```

Give the camera a couple seconds to start up.

Start a loop to take pictures repeatedly.

Wait 3 minutes.

When the camera has finished taking images, you can press Ctrl-C to end the program.

> NOTE This program saves the images in the folder where the program is being run. You should make a folder for your time-lapse project and run the program from that folder.

Next you'll need to combine the images into a video. You can use an application called `mencoder` to turn images into a movie. Install it like this:

```
pi@raspberrypi ~ $ sudo apt-get install mencoder
```

Then you'll create a simple text file that contains all the names of the images you want to combine. You can use the list command (`ls`), select all the files ending in .jpg, and output the list to a text file:

```
pi@raspberrypi ~ $ ls image*.jpg > list.txt
```

Next use `mencoder` to combine all the individual images into a time-lapse movie. This example makes a movie called TimeLapseMovie.avi:

```
pi@raspberrypi ~ $ mencoder -nosound -ovc lavc -lavcopts
➥ vcodec=mpeg4:aspect=16/9:vbitrate=8000000 -vf scale=1920:1080 -o
➥ TimeLapseMovie.avi -mf type=jpeg:fps=24 mf://@list.txt
```

When it's done, you can watch the movie using OMXPlayer:

```
pi@raspberrypi ~ $ omxplayer TimeLapseMovie.avi
```

You can read more online about the mencoder options available.

Raspberry Pi robot

The Raspberry Pi can readily be turned into a robot by adding servomotors and sensors using the Pi's GPIO capabilities. The Pi can be programmed in Python to make decisions, gather input from sensors, and control servomotors to interact with the world.

Although you could assemble you own robot from scratch, there are some Raspberry Pi robot kits that can make it a lot easier. For this project, we'll discuss using the GoPiGo kit from Dexter Industries. It's an affordable, well-engineered kit that within a few hours will let you have your Pi moving around under your control. You can add an ultrasonic sensor (detects objects in front of it) and write a Python program to make your Pi GoPiGo robot navigate the room autonomously (on its own) using the same `if`/`else` statements you learned earlier.

We'll cover the basic steps for building the GoPiGo (you can read the full set of instructions online at www.dexterindustries.com/GoPiGo):

1. Build your GoPiGo robot following the online instructions. Connect your Raspberry Pi to the robot: it fits upside down on top of the GoPiGo board. The Pi communicates to the GoPiGo board through the GPIO pins.
2. Insert the GoPiGo SD card into your Pi. The SD card contains a custom distribution of Raspbian. Connect your Pi to a keyboard, mouse, USB Wi-Fi adapter, and TV or monitor. Later you'll be able to connect to your GoPiGo remotely from another computer. Power it up using the provided battery pack. Boot up your Pi, and connect the GoPiGo to your wireless network.
3. Set up your computer to remotely access your Pi from another computer. This means you'll be able to see your Raspbian desktop from another Windows or Mac computer in your home. To do so, you use software called VNC. You need to install VNC Server on your

Raspberry Pi and then install VNC Client on your computer. There are some great tutorials on how to do this, such as the one on the Adafruit website. Go to https://learn.adafruit.com, and search for "installing VNC."

4. Using VNC, connect to your Raspberry Pi from your home Windows or Mac computer. Once you're sure VNC Server and Client are working properly, you can disconnect your Pi from the monitor, keyboard, and mouse, leaving only the USB Wi-Fi adapter plugged in. Your GoPiGo is ready to move!
5. From the VNC Client on your Windows or Mac, open LXTerminal on your Pi. Change directories to the GoPiGo Python folder on the desktop using the `cd` command:

```
cd Desktop/GoPiGo/Software/Python/
```

Run the GoPiGo test controller Python program:

```
sudo python basic_test_all.py
```

After the program starts, you can use these keys to move your GoPiGo around the room:

w	Move forward
a	Turn left
s	Move back
d	Turn right
x	Stop
t	Increase speed
g	Decrease speed

Excellent! You've made your Raspberry Pi into a robot. Add sensors and make programs to navigate around a room, or attach a Pi camera and stream video to another computer so you can see what your Raspberry Pi sees.

Index

Symbols

_ (underscore) 44
: (colon) 109, 112, 141
. (dot notation) 128
& (ampersand) 233
== (equality operator) 106, 142

Numerics

3.5 mm audio/video port 248–250

A

-a option 275
addition (math) 35–37
additive color 187
ampersand (&) 233
and operator 130
anode, defined 164
append method 224
as keyword 169
ASCII art 98
assignment operator 42
audio formats 207–208

B

Blinky Pi project
  breadboards
    electrical circuitry and 160–161
    holes in 159–160
    overview 158–159
  circuit for
    adding LED 164
    adding more LEDs 171–173
    connecting jumper from GPIO pin 163
    connecting resistor 164–166
    overview 161–166
  GPIO pins
    breaking out to breadboard 155–156
    overview 153–155
  overview 151–153
  program for
    adding more LEDs 173–174
    loading libraries 169
    main program loop 170
    overview 166–168
    running 168
    setting up GPIO pin for output 170
    troubleshooting 168–169
Boolean logic 108
booting
  defined 19
  issues 241
breadboards
  breaking out GPIO pins to 155–156
  circuit for Light Up Guessing Game
    adding jumper to ground 186–188
    adding resistors 185–186
    connecting GPIO jumper wires 183–184
    connecting RGB LED 182–183
    sketch 180–182
  connecting Pi to 179
  electrical circuitry and 160–161

breadboards *(continued)*
finding holes on 179–180
holes in 159–160
overview 158–159
bugs, defined 81
buttons
connecting in DJ Raspi project 210, 212–214, 218
connecting jumper wires to 215, 218

C
Camera Serial Interface. *See* CSI
capitalize method 78
cases 6
cathode, defined 164
central processing unit. *See* CPU
colon (:) 109, 112, 141
command-line mode 23
commenting code 73–75
common cathode RGB LEDs 182
comparison operators 108
component video input 256
computer vision 205
concatenation 80
conductance, defined 161
CPU (central processing unit) 7
CSI (Camera Serial Interface) 250–252
current, defined 160

D
datetime module 115
debugging 81–83
desktop, booting to 24–26
digital visual interface. *See* DVI
division (math) 37–38
DJ Raspi project
audio formats 207–208
circuit for
adding jumper to GPIO pin 216, 218
adding mini pushbutton 212–214, 218
adding resistor 215–216, 218
connecting jumper wire to button 215, 218
sketch 211–212
MP3 format 208–209
overview 204–207
program for
building list of sound files with os library 227–228
creating functions 231–234
getting length of list 226
getting list of sounds 221–225
getting value of item stored in list 225–226
initializing buttons 220–221
overview 218–220
playing sound when button is pressed 228–231
testing 234–235
troubleshooting 209–210, 235
wiring button 210
dot notation 78, 128
DVI (digital visual interface) 240
DVI port devices 16

E
electricity, defined 160
elif statements 129–130
else statements 127
equality operator (==) 106, 142
ethernet 6
Ethernet port 253
exponents 38–39

F
File Manager 27
fileinput module 115
fim program 274
flags, command-line 209
floating-point numbers 40
flow diagrams
overview 124–126
translating into code 131–133
for loop 228
functions
creating 133–138
creating for DJ Raspi project 231–234
defined 41
left cave exploration in Raspi's Cave Adventure 138–139
methods vs. 78, 128

right cave exploration in Raspi's Cave Adventure 139–141
troubleshooting 141–142
when to use 194

G
GoPiGo kit 283
GPIO pins 17
adding jumpers to 216, 218
breaking out to breadboard 155–156
connecting jumper wires for Light Up Guessing Game 183–184
defined 153
overview 153–155
purpose of 176
setting up pins for RGB LEDs 190–194
GPIO.cleanup() command 168
GPU (graphics processing unit) 143
GUI (graphical user interface) mode 23

H
Halloween heads project 279–280
hardware
cases 6
HDMI port
connecting TV or monitor 14–15
DVI port devices 16
overview 13–14
overview 4–6
ports 17
power supply 17
SD cards
inserting card in slot 12
NOOBS on 12
overview 11–12
portability of 13
replacing cards 12–13
system on a chip 7–8
USB ports
connecting keyboard 9–10
connecting mouse 10
overview 8–9
wireless keyboard and mouse combination 10
hashtag comments 73–75
HDMI port
connecting TV or monitor 13–15
defined 6
DVI port devices 16
overview 13–14

I
IDLE (Integrated DeveLopment Environment)
creating programs 54–56
overview 28–29, 33–34, 53–54
saving programs 56
if statements
in Norwegian Blue Guessing Game 105–109
using in loops 113
import statements 116
index, list 225
input
defined 6
getting from player 101–105
handling unexpected
and operator 130
elif statements 129–130
not operator ??–131131
or operator 129–131
overview 127–129
input function 75–76, 102, 126
Integrated DeveLopment Environment. *See* IDLE

J
jumper wires 152

K
keyboard
connecting to USB port 9–10
wireless 10

L
Leafpad 57–60
LEDs (light-emitting diodes) 151
legacy boards
Raspberry Pi 1 Model B 256–258
Raspberry Pi 1 Model B+ 258
legs, defined 164, 180

len() function 46, 226
libraries
  loading 169
  using in programs 115–116
Light Up Guessing Game
  breadboards
    connecting Pi to 179
    finding holes on 179–180
  circuit for
    adding jumper to ground 186–188
    adding resistors 185–186
    connecting GPIO jumper wires 183–184
    connecting RGB LED 182–183
    sketch 180–182
  overview 176–179
  program
    guessing game logic 197–198
    main game loop 195–197
    overview 188–190
    play again logic 198–200
    playing game 200
    setting up GPIO pins for RGB LED 190–194
    troubleshooting 200–201
  RGB LEDs 180
light-emitting diodes. *See* LEDs
list-comprehension feature 228
live streaming video 143–145
Livestreamer 143
loops
  using if statements in 113
  while loops
    breaking out of 113–114
    overview 110–113
    troubleshooting 114–115
lower method 78

M
math module 115
mathematical operators
  adding and subtracting 35–37
  exponents 38–39
  multiplying and dividing 37–38
  order of operations 48–50
  remainders 38
  square roots 39
memory 8
  See also SD cards
mencoder program 282
methods, functions vs. 78, 128
microSD cards 12
Minecraft Pi 85–88
miniSD cards 12
MIT (Massachusetts Institute of Technology) 118
monitors
  component video input 256
  connecting to HDMI port 14–15
  identifying ports 254
  RCA port 254–255
  VGA port 255–256
monitors, checking connection to 240–241
mouse
  connecting to USB port 10
  wireless 10
MP3 format 207–209
multiplication 37–38

N
nano text editor 59
negative power bus 218
NOOBS (New Out of the Box Software) 12, 242
Norwegian Blue Guessing Game
  getting player input 101–105
  if statements 105–109
  overview 91–94
  using libraries to generate random numbers 115–116
  welcome message and instructions 94–100
  while loops
    breaking out of 113–114
    overview 110–113
    troubleshooting 114–115
not operator 130–131

O
-o switch 209–210
Ogg format 207

OMXPlayer 208, 280
operators
  adding and subtracting 35–37
  comparison 108
  exponents 38–39
  multiplying and dividing 37–38
  order of operations 48–50
  remainders 38
  square roots 39
or operator 129–131
OS (operating system) 19, 227
os module 227, 230
output, defined 6

P
Pi NoIR module 252
Pi Store 29–30
picamera module 282
PIR (passive infrared) 279
ports 17
  3.5 mm audio/video port 248–250
  Camera Serial Interface 250–252
  defined 9
  Ethernet port 253
  overview 246
  TV/monitor
    component video input 256
    identifying ports 254
    RCA port 254–255
    VGA port 255–256
positive power bus 215, 218
power supply 17
print function 50
  Python 50–51
  troubleshooting using 200
PWM (pulse width modulation) 188
Python
  creating programs 54–56
  IDLE 28, 33–34, 53–54
  mathematical operators
    adding and subtracting 35–37
    exponents 38–39
    multiplying and dividing 37–38
    order of operations 48–50
    remainders 38
    square roots 39
  print function 50–51
  saving programs 56
  troubleshooting 51–52
  type checking 40–41
  using text editors 54
  variables
    box analogy 47
    changing value of 46–50
    creating and assigning values 42
    defined 41–42
    displaying values 42–45
    naming 43–44
    reassignment of 48
    strings in 45–46

Q
quotation marks 45

R
RAM (random access memory) 8
randint tool 116
random module 115
random number generation 115–116
Raspberry Pi
  cases 6
  hardware overview 4–6
  HDMI port
    connecting TV or monitor 14–15
    DVI port devices 16
    overview 13–14
  overview 4
  Pi Store 29–30
  ports 17
  power supply 17
  powering on checklist 18–19
  Raspberry Pi 1 Model B 256–258
  Raspberry Pi 1 Model B+ 258
  Raspbian operating system
    applications on 26
    booting to desktop 24–26
    configuring 21–24
    files and folders 26–27
    IDLE 28–29
    installing 19–21

Raspberry Pi *(continued)*
SD cards
inserting card in slot 12
NOOBS on 12
overview 11–12
portability of 13
replacing cards 12–13
system on a chip (SoC) *7–8*
updating 166
USB ports
connecting keyboard 9–10
connecting mouse 10
overview 8–9
wireless keyboard and mouse combination 10
Raspi's Cave Adventure
flow diagrams
overview 124–126
translating into code 131–133
functions
creating 133–138
left cave exploration 138–139
right cave exploration 139–141
troubleshooting 141–142
handling unexpected input
and operator 130
elif statements 129–130
or operator 129–131
overview 127–129
left cave 124
overview 121–124
right cave 124
raspistill program 252
RCA port 254–255
refactoring, defined 137
remainders 38
remove method 224
resistance, defined 161
resistors
adding for DJ Raspi project 215–216, 218
connecting for Light Up Guessing Game 185–186
purpose of 161
RGB LEDs
connecting to breadboard 182–183
overview 180
robot project 283–284

S

Scratch, overview 118–119
SD cards
inserting card in slot 12
NOOBS on 12
overview 11–12
portability of 13
reformatting 241–243
replacing cards 12–13
SDFormatter software 242
setmode function 170
Silly Sentence Generator 3000 69–71, *73–77*, 79–88
commenting code *73–75*
creating program 69–71
debugging 81–83
input function *75–76*
joining strings
building sentence 80–81
overview *77–79*
using multiple inputs *79*
Minecraft Pi 85–88
overview *67–69*
printing to screen 83–84
saving program *71–73*
SoC (system on a chip) *7–8*
SparkFun website 280
square brackets 223
square roots 39
streaming video 143–145
string literals *97*, 267
string methods 128
strings
joining
building sentence 80–81
overview *77–79*
using multiple inputs *79*
storing in variables *45–46*
subliminal messages 89
subtraction (math) *35–37*

sudo command 168, 200
switches, command-line 209
system on a chip. *See* SoC

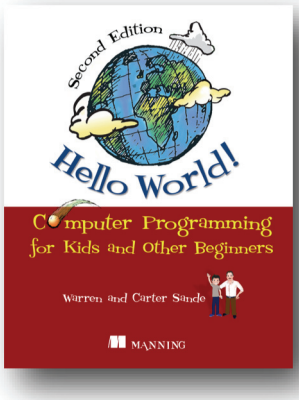
T
TAAG (text-to-ASCII art generator) 99
time-lapse photography project 281–283
triple double quotes 97
troubleshooting
  checking monitor connection 240–241
  checking power 239–240
  DJ Raspi project 209–210, 235
  functions 141–142
  incomplete booting 241
  Light Up Guessing Game program 200–201
  Python 51–52
  reformatting SD card 241–243
  searching online for help 243
  while loops 114–115
TV connections
  component video input 256
  connecting to HDMI port 14–15
  identifying ports 254
  RCA port 254–255
  VGA port 255–256
type checking 40–41

U
underscore (_) 44
upper() method 78, 129
USB ports
  connecting keyboard 9–10
  connecting mouse 10
  defined 6
  overview 8–9

V
variables
  box analogy 47
  changing value of 46–50
  creating and assigning values 42
  defined 41–42
  displaying values 42–45
  naming 43–44
  reassignment of 48
  strings in 45–46
VGA (video graphics array) 241
VGA port 255–256
video
  live streaming 143–145
  playing videos 142–143
voltage, defined 160

W
WAV format 207
while loops
  breaking out of 113–114
  overview 110–113
  troubleshooting 114–115
  using if statements in 113
whitespace 36
wireless keyboard/mouse 10

MORE TITLES FROM MANNING

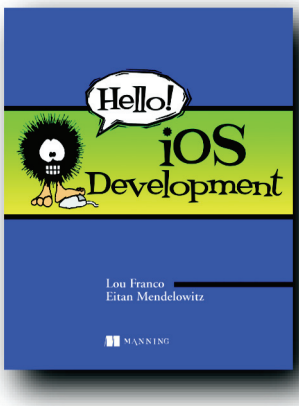


Hello World!
Second Edition
by Warren Sande and Carter Sande

ISBN: 9781617290923
464 pages
$39.99
December 2013

Hello App Inventor!
by Paula Beer and Carl Simmons

ISBN: 9781617291432
360 pages
$39.99
October 2014



Hello! iOS Development
by Lou Franco and Eitan Mendelowitz

ISBN: 9781935182986
344 pages
$29.99
July 2013

For ordering information go to www.manning.com

PYTHON / HARDWARE

Hello Raspberry Pi!™

Ryan Heitz

The Raspberry Pi is a small, low-cost computer invented to encourage experimentation. The Pi is a snap to set up, and using the free Python programming language, you can learn to create video games, control robots, and maybe even write programs to do your math homework!

Hello Raspberry Pi! is a fun way for kids to take their first steps programming on a Raspberry Pi. First, you discover how to set up and navigate the Pi. Next, begin Python programming by learning basic concepts with engaging challenges and games. This book gives you an introduction to computer programming as you gain the confidence to explore, learn, and create on your own. The last part of the book introduces you to the world of computer control of physical objects, where you create interactive projects with lights, buttons, and sounds.

What's inside

- Learn Python with fun examples
- Write games and control electronics
- Use Raspberry Pi to control lights and sounds
- Loaded with programming exercises

To use this book, you'll need a Raspberry Pi starter kit, keyboard, mouse, and monitor. No programming experience needed.

Ryan Heitz is a teacher, programmer, maker, father, and big kid. He specializes in teaching kids to code in a fun and engaging way.

"Very well written and inspiring, for both kids and teachers!"
—Dr. Christian Mennerich
Devoxx4Kids Team, Germany

"A fantastic resource for young programmers. Ryan Heitz does a great job walking readers through examples ... I wish this book had been available when I first started to program!"
—Nathan Sperry, student
Thomas Jefferson High School for Science and Technology

"This book makes coding easy and fun to learn."
—Matthew Giblin, age 13
Bradenton Preparatory Academy
Dubai

"A fantastic overview of the truly remarkable Raspberry Pi. My daughter Grace and I were thoroughly engrossed while working through the examples."
—Dan Kacenjar, Wolters Kluwer

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/hello-raspberry-pi

ISBN-13: 978-1-61729-245-3
ISBN-10: 1-61729-245-1

MANNING

US $24.99 | Can $28.99