



Community Experience Distilled

Raspberry Pi Projects for Kids

Second Edition

Leverage the power of programming to use the Raspberry Pi
to create awesome games

Daniel Bates

[PACKT] open source*
PUBLISHING community experience distilled

www.it-ebooks.info

Raspberry Pi Projects for Kids

Second Edition

Leverage the power of programming to use the
Raspberry Pi to create awesome games

Daniel Bates

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Raspberry Pi Projects for Kids

Second Edition

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: March 2014

Second edition: April 2015

Production reference: 1240415

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78528-152-5

www.packtpub.com

Credits

Author

Daniel Bates

Project Coordinator

Akash Poojary

Reviewers

Ed Baker

Colin Deady

Ian McAlpine

Proofreaders

Simran Bhogal

Safis Editing

Commissioning Editor

Priya Singh

Indexer

Tejal Soni

Acquisition Editors

Harsha Bharwani

Sonali Vernekar

Graphics

Sheetal Aute

Content Development Editor

Nikhil Potdukhe

Production Coordinator

Arvindkumar Gupta

Technical Editor

Rohith Rajan

Cover Work

Arvindkumar Gupta

Copy Editors

Sonia Cheema

Merilyn Periera

About the Author

Daniel Bates is a computer science researcher at the University of Cambridge. His day job involves inventing designs for future mobile phone processors and when he gets home, he likes playing games or working on one of his coding projects (or both!). Daniel has been a volunteer for the Raspberry Pi Foundation since 2011 and is enthusiastic about introducing new people to computing. He has previously written *Instant Minecraft: Pi Edition Coding How-to* and *Raspberry Pi Projects for Kids* (First Edition), both published by Packt Publishing.

About the Reviewers

Ed Baker graduated with a BSc in physics from Imperial College, London, in 2007 and somehow ended up working in the Entomology Department of the Natural History Museum shortly after. His work focuses on how technology, both hardware and software, can improve the way research is performed, from field data collection to final publication. Outside the technology world, he is a specialist on stick insects, cockroaches, and grasshoppers.

Recently, his work has focused on automated acoustic and environmental monitoring, protocols for sensor networks, and starting the biodiversity technology company <http://www.infocology.co.uk>.

Ed's first book, provisionally titled *Arduino for Biologists*, will be published in 2015 with Pelagic Publishing.

I would like to thank Philippa for believing that the writing and 'tinkering' would bring reward in the end.

Colin Deady's career in IT started in the late 1990s, when he discovered software testing ("they want me to break it?"), having previously fallen in love with computers when his parents brought him and his brother a ZX81 and ZX Spectrum+ in the 1980s and later an Amiga 1200 in the early 1990s. With over 15 years of experience in testing, he works as a Technical Test Manager, emphasizing the benefits of test automation and extolling the virtues of agile using Kanban and Behavior-Driven Development to great effect: define behaviors, then test early, test often. He combines BDD with a fix early, fix often approach that he terms Zero Known Defects.

Colin was one of the technical reviewers for Tim Cox's excellent *Raspberry Pi Cookbook for Python Programmers*, by Packt Publishing, and has written several articles for The MagPi, a community magazine for the Raspberry Pi. He has also reviewed and edited many more, building up extensive knowledge on this tiny platform.

He currently runs a blog related to all things Raspberry Pi, which can be found at www.rasptut.co.uk.

Ian McAlpine was first introduced to computers when he used his school's Research Machines RML 380Z and his physics teacher's Compukit UK101. This was followed by a Sinclair ZX81 and then a BBC Micro Model A, which he still has to this day. The interest he has in computers resulted in an MEng degree in electronic systems engineering from Aston University and an MSc in information technology from the University of Liverpool. Ian is currently a product expert in the Business Intelligence and Analytics Competence Center at SAP Labs in Vancouver, Canada.

The introduction of the Raspberry Pi rekindled his desire to "tinker" but also provided him with an opportunity to give back to the community. Ian is also a very active volunteer, who works for The MagPi, a monthly magazine for the Raspberry Pi community, which you can read online or download for free from www.themagpi.com. He also holds an amateur radio license (callsign VE7FTO) and is a communications volunteer for his local community emergency management office. He was a technical reviewer for *Raspberry Pi Cookbook for Python Programmers*, by Packt Publishing.

I would like to thank my darling wife, Louise, and my awesome kids, Emily and Molly, who've allowed me to disappear into my office and have, consequently, trained our dog to fetch me!

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	v
Chapter 1: Getting Started with Raspberry Pi	1
Materials needed	2
Power supply	3
Storage	4
Input	5
Video	6
Network	7
Preparing the SD card	8
Starting up the Raspberry Pi	11
Using your Raspberry Pi	14
The command line	15
Updating and installing new software	17
Other uses of the Raspberry Pi	18
Troubleshooting common issues	19
Summary	20
Chapter 2: Animating with Scratch	21
Scratch	21
Hello world!	23
Code tour	24
Some more interesting movements	25
Setting the scene	26
Another way to animate	28
Interactive animation	29
Variables	30
Movement	32

Keeping count	33
If-then-else	34
Summary	36
Chapter 3: Making Your Own Angry Birds Game	37
Creating a character	38
Creating a level	40
Moving the character	41
Initialization	42
Moving the character with the keyboard	42
Launching the character!	44
Flight	45
Adding physics	46
Gravity	46
Bouncing	46
Ending the game	47
Scoring	48
Extensions	51
Summary	52
Chapter 4: Creating Random Insults	53
Python	53
Python programming	55
The program we're going to use to generate phrases	55
Lists	56
Adding randomness	57
Creating phrases	58
Making mischief	59
Dictionaries	59
Loops	60
Conditionals	61
Functions	62
Complete code listing	64
Summary	65
Chapter 5: Testing Your Speed	67
Materials needed to make your own controller	67
Creating the game controller	68
The controller base	69
Adding buttons	69
Connecting to the Raspberry Pi	71

Coding the game	72
Random behavior	73
Using the controller	74
Adding a time limit	75
Bringing it all together	77
Complete code listing	79
The keyboard version	80
What's next?	81
Summary	82
Chapter 6: Making an Interactive Map of your City	83
Hello world!	84
Tkinter	84
Writing the program	84
Getting a map	87
No Internet? No problem!	87
Google Maps	87
Generating the address	88
Downloading an image	89
Using an image	90
Adding markers	92
Detecting mouse clicks	92
Reacting to mouse clicks	93
Adding labels	94
Basic labels	94
Pop-up windows	95
Code listing	98
Extensions	100
Layout	101
Additional widgets	101
Checkbutton	101
Frame and LabelFrame	102
Listbox	102
Menu	102
Menubutton	103
Message	103
OptionMenu	103
Radiobutton	104
Scale	104
Spinbox	104
Summary	105

Chapter 7: Building Beats with Sonic Pi	107
Sonic Pi	107
Getting started with Sonic Pi	108
Creating a tune	109
New sounds	111
A real tune	112
Adding rhythm	115
Bass line	116
More fun	119
Code listing	120
Summary	122
Index	123

Preface

The Raspberry Pi is a credit card-sized computer designed to make computing accessible to all. With the trend towards making computers easier and easier to use, the art of programming has been in decline. Programming is a powerful tool that lets us tell the computer exactly what we want to do. In much the same way as we use a hammer or screwdriver to help us with a physical task, we can use programming to help us with a mental task. The Raspberry Pi exposes programming software to make it as easy as possible to get started.

After introducing the Raspberry Pi computer and showing you how to set it up, this book will guide you through six separate mini-projects. Each project is fun, visual, and has plenty of scope for personalization. By the end of this book, you will understand and be able to use three different programming languages, and will be able to use them to build creative programs of your own.

What this book covers

Chapter 1, Getting Started with Raspberry Pi, shows you what Raspberry Pi is and how you can get one set up and ready to use.

Chapter 2, Animating with Scratch, introduces the Scratch programming language and uses it to create simple (and not-so-simple) animations.

Chapter 3, Making Your Own Angry Birds Game, teaches you how to make your very own computer game using the Scratch programming language.

Chapter 4, Creating Random Insults, explores how random funny phrases can be generated using the Python programming language.

Chapter 5, Testing Your Speed, helps you to connect electronic components to your Raspberry Pi to create a physical game controlled by your computer code written in Python.

Chapter 6, Making an Interactive Map of Your City, teaches you more about Python and shows you how to access Google Maps to create a personal map of your area.

Chapter 7, Building Beats with Sonic Pi, introduces the Sonic Pi application and shows you how the programming concepts learned so far can be applied to the creation of music.

What you need for this book

All the projects in this book require Raspberry Pi and all the necessary peripherals (listed at the beginning of *Chapter 1, Getting Started with Raspberry Pi*). *Chapter 5, Testing Your Speed*, adds simple electronic components, listed at the beginning of that chapter. *Chapter 7, Building Beats with Sonic Pi*, requires headphones or speakers.

Who this book is for

This book is designed to help adults and children jump into creative coding using the Raspberry Pi. You will need patience, a sense of adventure, and a vivid imagination!

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text are shown as follows: "Be very careful when using the `sudo` command."

A block of code is set as follows:

```
def count(maximum):  
    value = 0  
    while value < maximum:  
        value = value + 1  
        print "value =", value
```


When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:


```
def count(maximum):  
    value = 0  
    while value < maximum:  
        value = value + 1  
        print "value =", value
```

Any command-line input or output are written as:

```
sudo apt-get upgrade
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Select **Raspbian** and click on **Install**."

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: https://www.packtpub.com/sites/default/files/downloads/15250S_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

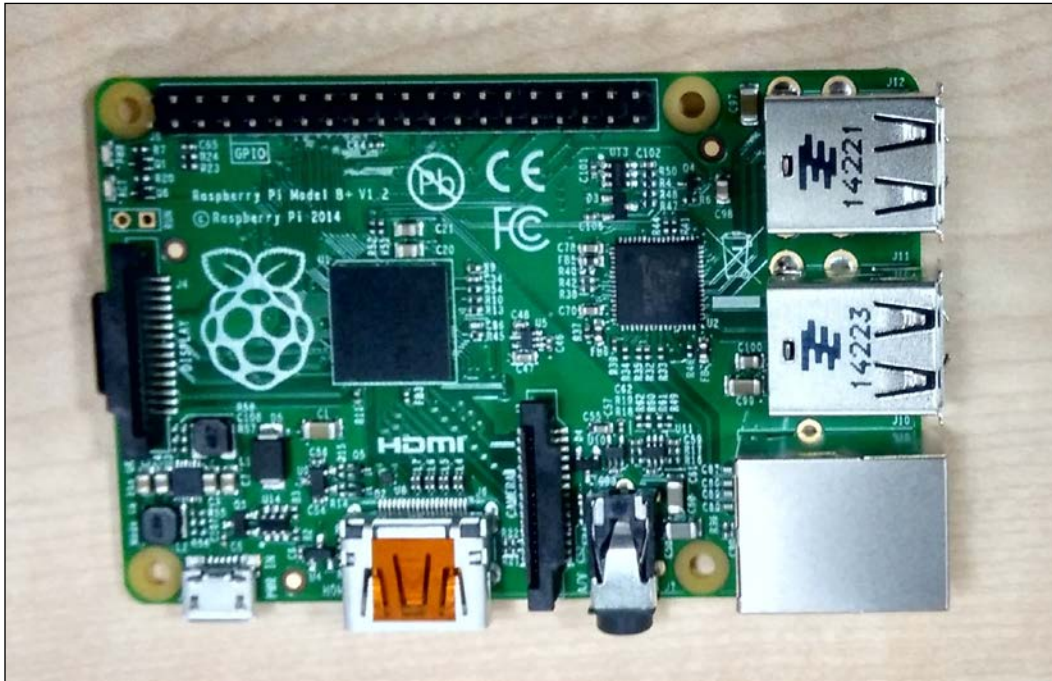
1

Getting Started with Raspberry Pi

In the mid-2000s, some of the staff at the University of Cambridge noticed that there were fewer students applying to study computer science each year, and that they didn't have very much experience. Something had to be done about this situation. The answer was the **Raspberry Pi**: a small, inexpensive computer which makes programming as accessible and as fun as possible. The idea is that students can play with the Raspberry Pi in their spare time, and in the process, learn valuable core computer science skills. Since its creation, many other groups have discovered how useful the Raspberry Pi can be, including schools, adults who want to brush up on their skills with technology, and electronics hobbyists.

This chapter describes how to get a Raspberry Pi computer up and running. Once this is done, the Pi behaves just like any other ordinary computer, and is capable of standard tasks, such as browsing the web and playing games. We will learn in later chapters that the Raspberry Pi is also capable of some things which ordinary computers can't easily do.

The following image shows an example of the Raspberry Pi that we will be using in the rest of the book:



Materials needed

Any model of the Raspberry Pi will work for the projects used in this book. The preceding image shows a Raspberry Pi Model B+, with four USB ports and a network connection. The Model A+ (with one USB port and no network connection) will also work, but a USB hub (which is described later) will be needed to allow both a keyboard and mouse to be used at the same time.

The Models A+ and B+ replace the older Models A and B, and have the same capabilities, but neater designs. You can identify the Plus models by looking at the mounting holes surrounded by metal. The Plus (+) models have four mounting holes in a rectangle, whereas the previous models have two or zero mounting holes. In the preceding image, two mounting holes are in the corners at the left end of the board, and the other two mounting holes are on the mid-right of the board.

The Raspberry Pi 2 looks almost the same as Model B+, but has a faster processor. This is not important for the projects in this book. Along with a Raspberry Pi computer, you will need other peripherals. In order to keep costs down, the Raspberry Pi was designed to work with devices that people already owned; you may find many of these components around your house already. Just make sure they're not in use before you take them!



http://elinux.org/RPi_VerifiedPeripherals is a useful website to check whether a particular device will work with the Raspberry Pi.

Power supply

The Raspberry Pi requires a micro-USB connection, which is capable of supplying at least 700 mA (or 0.7A) at 5V. Power supplies, which can give 1000 mA or more are available (and will be more reliable), but it must supply exactly 5 V. Most standard mobile phone chargers are suitable for this purpose, and have their capabilities written on them, so you can check. Do not attempt to power your Pi from the USB port of another computer or hub; they are often incapable of supplying the required current.



Storage

The operating system and files of Raspbian are stored on an SD card, which is similar to what you may find in a digital camera. The Raspberry Pi models A and B use a standard-sized SD card, while the A+, B+ and Pi 2 models use smaller microSD cards. You will need at least 4 GB of space (preferably 8 GB or more). The Raspberry Pi Foundation sells very affordable 8 GB SD cards with the operating system pre-installed, and these can be found at <http://swag.raspberrypi.org/>.

If you start with a blank SD card, you will also need a way of writing to it from another computer. Many computers have SD writers built in, but it is possible to buy USB dongles which do the job too.



Input

For inputs, we will use a USB keyboard and mouse:



Keyboard



Mouse

Video

We will use a monitor or a television with **HDMI** or **DVI** input, and a video cable connected from the Pi's HDMI port to the screen's input, as shown in the following figures. It is possible to connect to an older **VGA** or composite screen, but this is more complicated. Take a look at this website more information http://elinux.org/RPi_VerifiedPeripherals.

The following images show the display monitor and cables that you might use to connect the Pi to your monitor:



Monitor



DVI connector

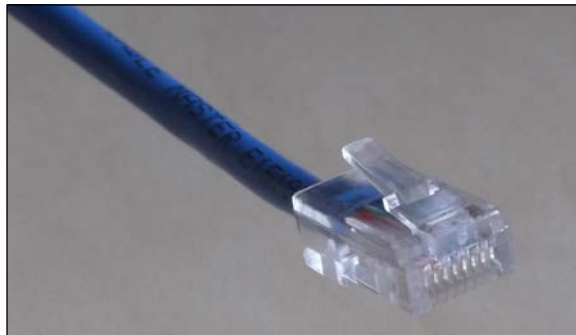


HDMI connector

Network

An Internet connection is not essential, but is very useful as it allows you to do more work directly on the Pi. The easiest approach is to use a wired Ethernet connection.

The following image shows an Ethernet connector that is used to connect the Pi to the Internet:



RJ-45 Ethernet Connector

It is also possible to use a USB Wi-Fi dongle – you will need a powered USB hub to provide additional USB ports, and you should check whether the dongle is compatible with a Linux operating system. The following image shows an USB hub (unpowered):



USB multi-port hub

You may also like to put your Raspberry Pi in a case to protect it, though this is certainly not necessary. There are many different companies which make different styles of cases, so choose one that suits you, or you could even make your own from Lego or cards!

Preparing the SD card

The first thing we need to do is put an operating system on the SD card using another computer. You can buy SD cards with software preinstalled, but doing it yourself guarantees to get you the latest updates, and is also a useful learning experience. These instructions assume that you are using a computer running Microsoft Windows or Mac OS X. If you are using another operating system, or if you are having difficulties in setting up the SD card, detailed instructions are available online at <http://www.raspberrypi.org/downloads>.

There is a *Troubleshooting* section at the end of the chapter if you get stuck.

The following steps show how you can install the OS on your Raspberry Pi:

1. Download the SD association's formatting tool, SD Formatter, from: http://www.sdcard.org/downloads/formatter_4/.
2. Download the latest version of the NOOBS (the offline installation) operating system collection from <http://www.raspberrypi.org/downloads>.

Insert the SD card into the SD card writer, as shown:

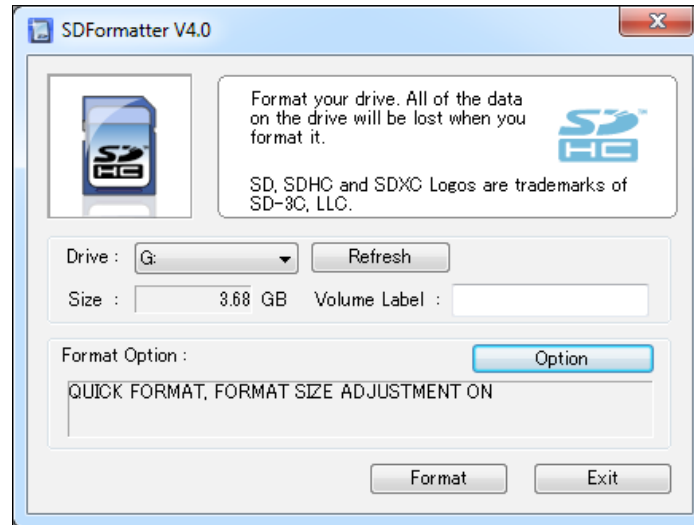


1. If the SD card writer is separate from your computer, plug it in.
2. Install and run SD Formatter. Select the SD card you just inserted, click on **Option**, then set **FORMAT SIZE ADJUSTMENT** to **ON** and click on **OK**. In this example, the SD card is shown as the **G** drive, but this will vary from computer to computer. Finally, click on **Format**.



Make absolutely sure that you have the right SD card selected.
If it isn't, all the data will be lost from the formatted card.

The following screenshot shows the SD Formatter software:



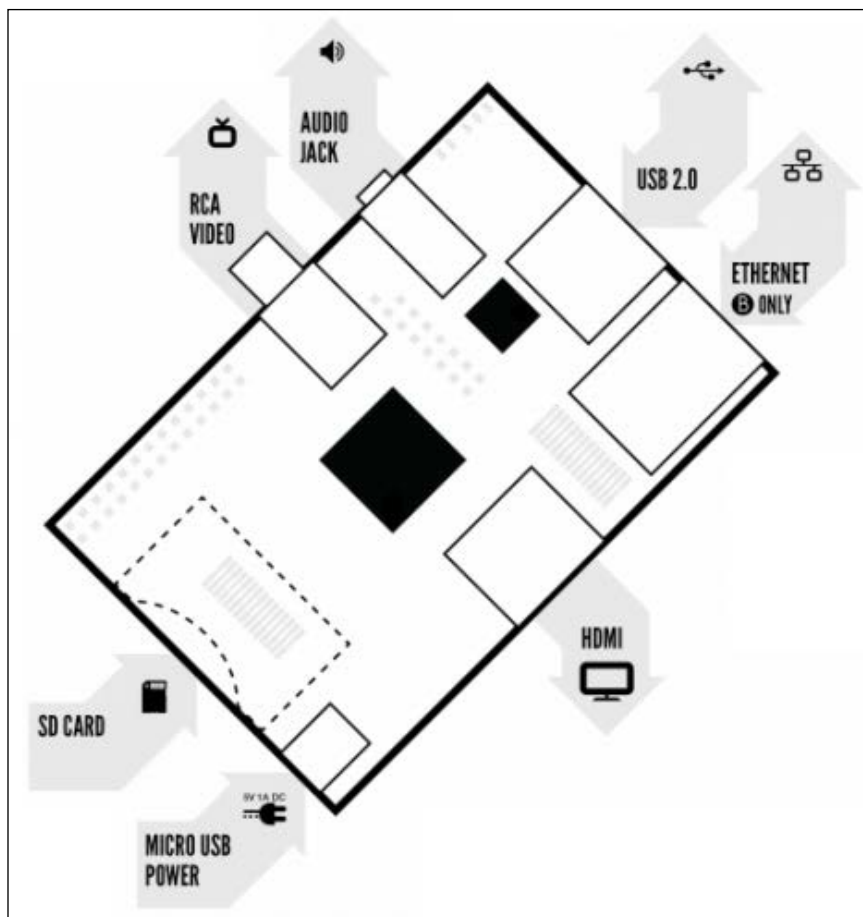
3. Extract the contents of the NOOBS zip file to the SD card. How this is done will vary depending on the software you have installed, but will typically involve double-clicking on NOOBS.zip, clicking on **Extract** or **Extract to...**, and selecting the SD card as the destination. There is a lot to extract, so this will take a few minutes to complete.
4. On your taskbar (close to the clock) click on **Safely remove/Eject** the SD card and take it out of the SD writer, as shown here:



Starting up the Raspberry Pi

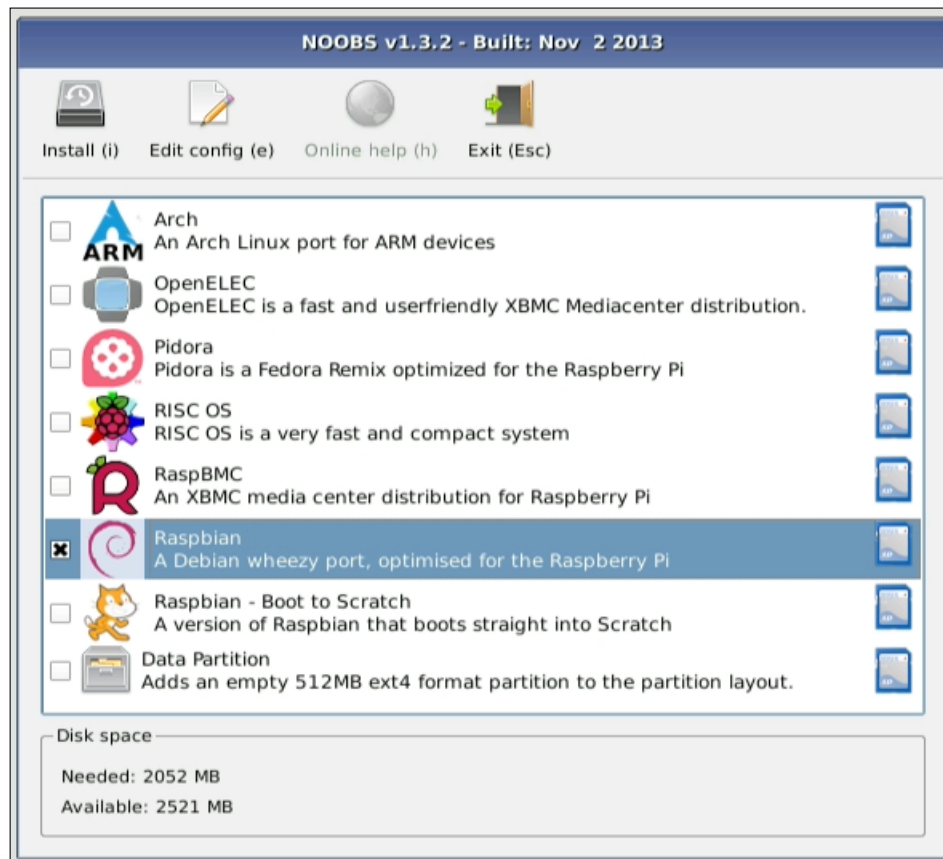
Now, we can prepare the Raspberry Pi to start for the first time. Place it securely on a desk or in a case. Make sure it is not in danger of falling on the floor, and do not rest it on top of the bag in which it comes. We can start up the Raspberry Pi by performing these steps:

1. Plug the SD card, screen, keyboard, and mouse into the Raspberry Pi. Also, plug in the Ethernet cable, if you have one, as shown here:



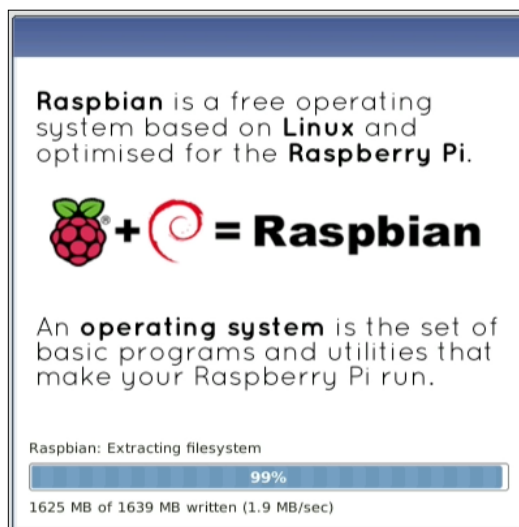
2. Plug the power cable into the Raspberry Pi. The red power light should come on, and the green activity light should flash occasionally.
3. If necessary, adjust the screen settings to display the images from the Raspberry Pi's input.

4. You should see a selection of operating systems for you to install (shown in the following screenshot), each with a short description. This book relies on you having Raspbian installed, so select **Raspbian** and click on **Install**. You can always come back and select a different operating system later; I will explain how you can do this in the next section.

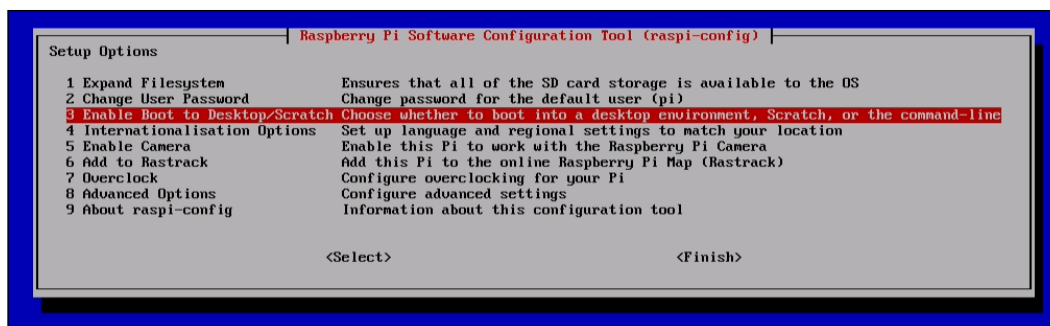


5. Wait! Operating systems are quite large, so installation will take a few minutes. You can sit back and read some of the tips shown to you, or read the next few steps in this book.

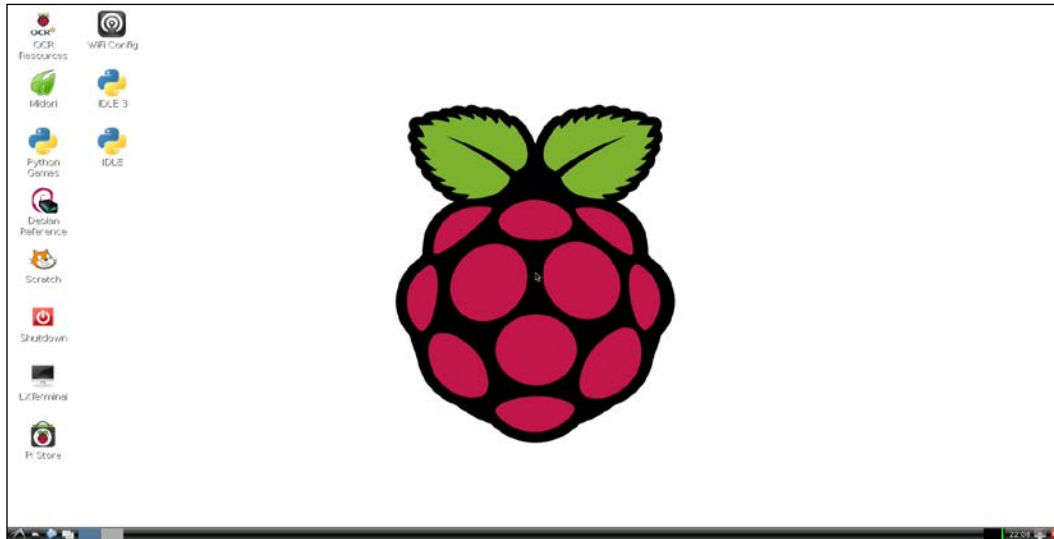
The following screen shows the Raspbian installation process, along with the tips shown while its being installed:



6. When the installation has finished, you should see a blue screen with a final list of options (shown in the following screenshot). This is the **Raspberry Pi Software Configuration Tool**. Most things should be set up the way we want them, but there are two useful settings to change. Select **Enable Boot to Desktop/Scratch** using the arrow keys and press *Enter*. Select the **Desktop Log in** option, and press *Enter*. You should now be back at the main menu. Next, select **Internationalisation Options**, and choose your preferred language and keyboard layout. Use the right arrow key to get to **Finish** and press *Enter*. You can return to this menu at any time by typing `sudo raspi-config` in a command line (refer to the next section for details):



7. After a minute or so, the Raspberry Pi should finish rebooting, and you should see the Raspberry Pi desktop, as shown in the following screenshot. This might look familiar to you: you can double-click on the icons to start programs, or select them from a menu. We will mainly be using Scratch, Python and Sonic Pi in this book, but take a minute to explore what's available to you. In particular, there are several **Python Games**: these are the sort of things that are possible to develop after a little programming practice:



Using your Raspberry Pi

Now that your Raspberry Pi is up and running, you'll want to know how to keep it working properly, and how to customize it to your needs.

The command line

Most of the time, it will be possible to do what you want to do using the mouse, by clicking on different parts of the screen, but at some point, you might find yourself needing to use the command line, as shown in the following screenshot:



The command line is a completely text-based way to control a computer, and can be used to do just about anything which can be done by clicking and more. It is available on almost all computers, but is usually hidden away. Some computer users prefer using the command line because they can type faster than they can click the mouse!

Here is a very quick overview of some common commands. Open a command line by double-clicking on the **LXTerminal** icon on the desktop, and try these out. You will need to press *Enter* to tell the Raspberry Pi that your command has finished. A longer introduction, including information on how to watch a movie in the command line, can be found online at <http://www.techradar.com/news/computing/pc/1161712>.

The following are few of the most important commands that are used in LXterminal:

- `ls` (list): This command lists directory contents (*directory* is Linux's word for a folder.) This will show you all of the files and directories available to you in the current directory.
- `cd <folderdirectory name>` (change directory). This command allows you to move into another directory so you can see its contents, in the same way that double-clicking on a directory icon moves you into that directory. You can move through multiple levels of directories in one go by separating the folder directory names with a `"/`, and you can go up to the parent directory (the directory that contains the current directory) using the special `..` directory name.
- `man <program name>` (open manual): This command will bring up lots of information about a particular program, including what it does and how to use it. Very useful if you forget how to use something! Try `man ls` to see some advanced information on the `ls` program we tried earlier, and press `Q` on the keyboard to quit. You can scroll through this information using the arrow keys or the *Space Bar*.
- `<program name>` (extra information): This is used to start the program, and optionally pass it some extra information. Try typing in `scratch` to start the Scratch program (we'll cover this in more detail in the next chapter), or if you are connected to the Internet, go to `epiphany www.raspberrypi.org` to open the Epiphany web browser and then go straight to the Raspberry Pi home page.



The *Tab* key can be used to automatically complete a word. Even if you have not completely finished typing in the name of a program or file or folder, try pressing *Tab*. If there is only one option available, which begins with the letters you have typed so far, the whole word will be completed for you. If there are multiple options (or none), nothing will change; you can press *Tab* again to show you a list of possibilities.

Updating and installing new software

If you want to install a program on your Raspberry Pi, you either need to download a version which is specifically for the Raspberry Pi, or use Raspbian's **package** system.

A package is a program or a part of a program, and many versions of Linux (including Raspbian) maintain a list of all the compatible packages, making it easy to keep all your software up to date. You can update to the latest version of this list if you have an Internet connection, by typing the following command in a command line:

```
sudo apt-get update
```



Be very careful when using the `sudo` command. It forces the Raspberry Pi to do exactly what you tell it to do, without checking to make sure that the command is sensible. The command is useful in situations like this, where we want to make changes to the installed programs, but it also allows you to delete essential files. Double-check your spelling before continuing.

You can search for available packages with keywords using the following command:

```
apt-cache search <keywords>
```

Try the following command, for example, to see a list of available free games. You could even try installing one (*xbubble* is good, for example):

```
apt-cache search game
```

The name of the package is the first word of the line, and you can install a package using the following command:

```
sudo apt-get install <package name>
```

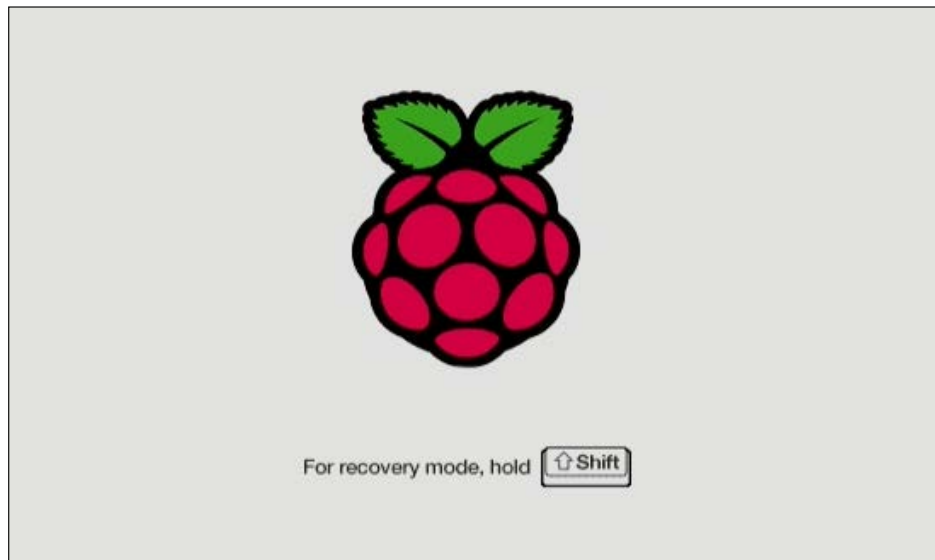
To update all the installed packages to the latest available version, type the following command:

```
sudo apt-get upgrade
```

Other uses of the Raspberry Pi

Although the Raspberry Pi was designed to get people interested in computing, its cost and power mean that it is also popular for other reasons. Since the Raspberry Pi is a general-purpose computer, it is capable of everything a traditional computer can do, just perhaps a little slower. There is a web browser (Epiphany), and word processors and web servers are available in the Pi that helps you with the basic computing needs. A common use for Raspberry Pi is as a media center, to watch films and view pictures.

There are many different operating systems included with the NOOBS package: if you have NOOBS installed, you can see them if you press *Shift* when the Raspberry Pi first starts to boot; you will get a screen similar to this:



This will take you back to the list of operating systems you saw earlier when you started your Raspberry Pi for the first time. Each operating system comes with a short description: there are a couple of different flavors of Linux, the very fast **RISC OS**, and two different media centers, **OpenELEC** and **RaspBMC**.

If you want to try out one of these operating systems, make sure you first back up all your data, as it will be erased when the new operating system is installed.

Troubleshooting common issues

One of the main strengths of the Raspberry Pi is its fantastic community. If you ever have any difficulties, consider stopping by the Raspberry Pi forums, which can be found at <http://www.raspberrypi.org/forum/>. Your question may have already been asked, and if not, there are thousands of enthusiastic Pi owners on hand to help. Some of the most common issues are covered here:

- *My Raspberry Pi doesn't boot* (only the red power light comes on): this suggests that the SD card was not written correctly. Try following the instructions again to copy an OS image onto your SD card, and if this fails, try a new SD card.
- *My Raspberry Pi randomly restarts by itself*: this is usually because the Pi is not receiving enough power. Double-check that your power supply is capable of supplying at least 700 mA (0.7A) at 5V. This should be written on the supply somewhere. Perhaps, try upgrading to a 1000 mA (1.0A) supply if you continue to have problems.

Also make sure that you do not have particularly power-hungry peripherals plugged into your Raspberry Pi. Some Wi-Fi dongles and keyboards with extremely bright LEDs can cause problems, for example.

- *I can't enter my password on the login screen*: nothing is shown when the password is entered (not even stars) to minimize the information that others can gain from seeing the screen. It is likely that the keys are still being recognized: try typing in the whole password blindly and pressing *Enter*.
- *The display does not fill my screen or extends beyond the edges*: this is because of the **overscan** settings. Many old televisions had cabinets which overlapped part of the screen, so images were given black borders to ensure that none of the picture was lost. Many modern monitors do not have this problem, however, so the black bars are just a nuisance. First, try enabling or disabling overscan by typing `sudo raspi-config` in a command line and selecting the appropriate option. If this still does not work, search the Internet for *Raspberry Pi overscan troubleshooting* for detailed guides.
- *I can't see anything at all on the screen*: if the Pi is definitely on, and the OK/ACT light is lit or flashing, try pressing 1, 2, 3, or 4 on your keyboard to select different video modes.
- *My keyboard is behaving strangely*: some wireless keyboards occasionally do not work as expected, for example, by registering double key presses or not noticing a key press at all. This should be fixed in the future by software updates, or you can always try another keyboard if you have one.

Summary

In this chapter, we learned how to connect a Raspberry Pi computer, write its operating system to an SD card, and initiate all its components. We learned that the Raspberry Pi is capable of everything a normal computer can do (and more), and that it is targeted at programming.

In the next chapter, we will use one of the provided programming languages, such as Scratch, to create our own animations.

2

Animating with Scratch

In this chapter, we're going to use a programming language called **Scratch** to create a simple animation. Along the way, we'll visit many of the main concepts of programming languages, so if you understand everything that you learn in this chapter, you will be well equipped to start writing programs of your own.

Scratch

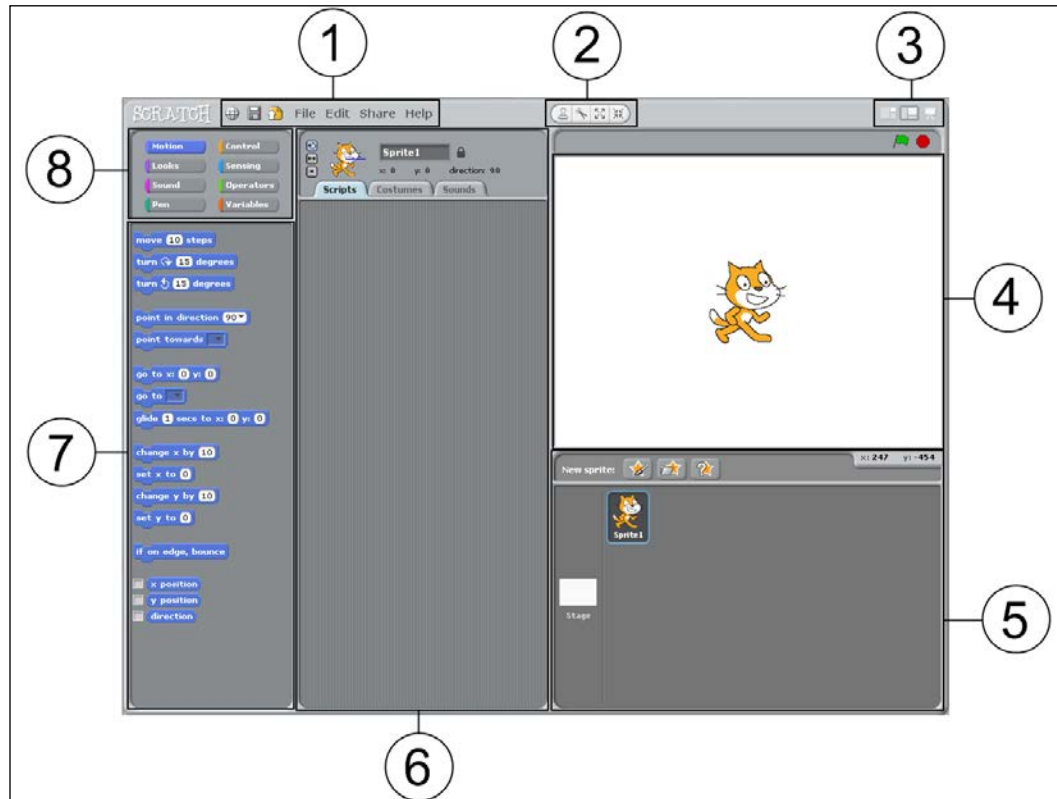
In this chapter, we will use Scratch to create our animation. Scratch is a programming language that has been specially designed so that you can make animations and games with ease. Version 1.4 of Scratch is pre-installed with the Raspbian OS but is also available on other computers. You can download it from <http://scratch.mit.edu/> if you ever want to run your programs away from your Raspberry Pi. Start up Scratch by opening **Menu** at the top of the screen, and navigating to **Programming**, and then **Scratch**.

Downloading the example code



You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

The following screenshot shows the layout of Scratch:



The following are its main sections. I'll mention the names of these sections throughout the next two chapters, so you might want to refer back to this page.

The following are the elements of Scratch, as shown in the preceding screenshot:

- **Menu (1):** This is where the options to save and load your projects are. If you ever want inspiration for projects, take a look at the provided examples by navigating to **File | Open | Examples**. Remember to save and back up your progress regularly!
- **Sprite controls (2):** Every picture in the game is called a sprite. These buttons allow you to copy, remove, grow, and shrink sprites. To use them, click on the button you want, and then click on the sprite you want to affect.
- **Screen layout (3):** Choose between a small Stage (see 4), a large Stage, and a fullscreen game. The small Stage is better for smaller screens as it allows more space for code.

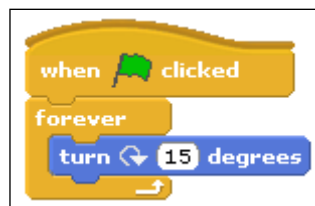
- **Stage (4):** This is where you will see the effects of all your programming.
- **Sprite list (5):** All the sprites in your project are shown here, and you can easily add new pictures or change existing ones in it.
- **Script area (6):** Each sprite has a number of scripts attached to it, and they are shown in this area. Each script is a short piece of code that controls how the sprite behaves.
- **Blocks (7):** Each block is a programming command that can be connected to other blocks (similar to a jigsaw) to create scripts. Drag a block into the script area to use it, and then drop it next to another block in the script area to join the two.
- **Block types (8):** The blocks are separated into eight different categories, each having different roles in your programs.

Hello world!

Let's create a very simple program to show how easy it is to produce a visible result.

1. From the **Control** section, drag a **when green flag clicked** block into the script area. Then drag a **forever** block so that it connects to the bottom of the **when green flag clicked** block. Finally, from the **Motion** section, drag a **turn 15 degrees** block into the middle of the **forever** block.
2. Once the blocks are connected, you can move them all around at the same time by clicking and dragging the topmost block. If you drag a block in the middle of a collection of blocks, you will get that block and all the blocks below it.
3. Click on the green flag present at the top-right corner of the screen to run the program.

The following screengrab graphically illustrates the preceding steps:



You should see the cat rotating. Your script should also be highlighted to show that it is active. You can change the rotation amount to any number you like to see the cat spin faster or slower—click on **15**, seen in the preceding code block, and type in a new number. You can even choose a negative number, and the cat will spin in the opposite direction. Click on the red stop sign in the top-right corner to stop your program.

Now, I'll describe how the Raspberry Pi understands your program and knows what to do. It understands that a script should start when the green flag is clicked on because this is the top block. As soon as this has happened, it moves on to the next block, **forever**. Everything inside the **forever** block will execute repeatedly until you tell it to stop. In this case, we have told the Raspberry Pi that we want to continuously rotate the cat, and this is what we see. You can see that no blocks can be attached to the bottom of the **forever** block. If an action keeps on going forever, no later commands will ever run.

Code tour

There are several types of code blocks available if you want to continue experimenting before we start working on the animation. A full description can be found online at http://info.scratch.mit.edu/Support/Reference_Guide_1.4. A quick tour of the code blocks is as follows:

- **Motion:** This allows us to control where a sprite is on the screen and which direction it is facing. Its options include rotating, moving to any position, and moving in the direction that the sprite is facing.
- **Control:** This allows us to choose when other blocks of code should run. In the preceding example, we saw how to decide when a script should start and how to repeat a block; however, it is also possible to execute a block only if a given condition is true.
- **Looks:** These enable us to decide what a sprite will look like. Each sprite can have multiple images or **costumes** associated with it, and these blocks can be used to switch between the two. It is also possible for the sprites to talk or change in size or color.
- **Sensing:** This enables us to allow a sprite to detect its surroundings.
- **Sound:** This enables us to play sound. You can add new sounds from the **Sounds** tab in the script area.
- **Operators:** These are simple mathematical functions, such as add and subtract. Note that some of the blocks are of different shapes; they show which blocks fit together and will be important later.

- **Pen:** This enables us to allow a sprite to draw a line to show where it has been.
- **Variables:** These allow us to give names to pieces of information so that they can be accessed from multiple places. We will go into more detail on this later in the chapter.

If you ever add a block which you no longer want, you can either drag it back to the **Blocks** area, or right-click on it and select **delete**.

Some more interesting movements

A rotating cat is fun, but isn't particularly interesting, is it? Let's see if we can do something a little better.

1. Drag a **move 10 steps** block from the **Motion** section and place it anywhere inside your existing **forever** block. The cat should now move in a circle when you click on the green flag, rather than just spinning on the spot.
2. Adjust the numbers in the **move** and **rotate** blocks until you are happy with the cat's movement. A larger number in the **move** block will make the circle larger and the motion faster. A smaller number in the **rotate** block will also make the circle larger, but this time it will take longer to complete a rotation. If the cat moves to a position you don't like, you can always drag it around on the Stage.
3. If you like the path the cat is taking, but think it is moving too quickly, you might like to try adding in a **wait 1 secs** block (from the **Control** section) inside your **forever** block, and reducing its number to something very small, such as 0.01. I find the following to be a good combination:



4. Now, cats don't usually move around in circles like this, so let's choose something a little more appropriate. At the top of the script area, click on the **Costumes** tab, then click on **Import**. You should see a whole selection of different images to use, including animals, people, and things.

5. Choose an image of something that you think would be more likely to move in the circular motion you created, such as a fish, bird, or an airplane.
6. Click on **OK** when you're happy with your selection, and you will be returned to the main Scratch screen.
7. You should now be able to see the image you chose, and also a couple of slightly different versions of the cat image.
8. We won't need the cat images anymore; remove them by clicking on the small **X** symbol next to each one.

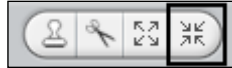
Setting the scene

We now have an image flying around in empty space. Let's add a background to make it look a bit better.

1. In the sprite list, on the left-hand side, you will find a special sprite called **Stage**. This serves as the background of the animation. Click on it, and the script area will update to show you information on the Stage. You should see that there is one available background simply called **background1**, and it is a white rectangle (if you don't see this, click on the **Backgrounds** tab at the top of the script area).
2. Just like we did before, click on **Import**, and choose a background that you think suits your sprite.
3. When you've finished, click on **X** next to the white background and you should be able to see your sprite and the new background on the stage. I chose an airplane as my sprite, so I've chosen a background which has some sky for it to fly around in:



4. As you can see, my plane looks like it has already crashed! I need to put it in a better position on the stage. If you also need to move your sprite to a different location, click on the sprite on the stage and drag it to a better position.
5. I also want to make my plane a little smaller. This can be done using the final button in the sprite controls section (as shown here). The button next to it can be used to make sprites larger.



6. Click on one of the buttons, and then click on your sprite repeatedly until it is the size you want it to be. You might want to adjust its position again when you are happy with its new size. Here's an illustration of my stage now:



As you can see, I've moved the plane into the sky, and made it smaller so that it appears to be further away.

Another way to animate

Remember how we earlier removed two slightly different cat images? They were there to allow a different way of animating, that is, switching between different images, or **costumes**. The following are the steps needed to switch between images and costumes:


1. At the top of the sprite list, click on the middle button. It says **Choose new sprite from file** when you hover your mouse cursor over it.
2. Find an image you like and that has multiple versions of it available, such as **crab1-a** and **crab1-b**. Choose the former, and click on **OK**.
3. Now, go into the **Costumes** tab and **Import** the second version of the same sprite, but this time, do not remove the costume that you already had. If your chosen sprite has more than two versions of it, repeat this process until you have all of them. You should see a screenshot similar to this one when you're done:



4. Now that we have all the costumes we want, let's write a script that cycles through them to create an animation. Click on the **Scripts** tab at the top of the script area and build the following script in it. You'll notice that your script for the first sprite isn't here. Each sprite has its own collection of scripts that decides how it should behave. You can always return to see a particular sprite's scripts by clicking on that sprite in the sprite list:



5. The **next costume** code block can be found in the **Looks** section of the blocks area. Click on the green flag to see what your animation looks like – you may want to adjust the time taken between switching costumes, so as to make your animation look better.

 The background can also be animated using this technique – some of the backgrounds have multiple versions, and the stage can have its own collection of scripts to change the way it looks.

6. Finally, in the same way as we did for our first sprite, let's choose a sensible size and position for this second sprite. Drag it to a better position on the stage, and use the grow or shrink buttons to change its size.

Both of the animation methods we have met so far can be combined. It is possible to change a sprite's costume, and move it around the screen. This is as easy as adding a second script to a sprite. If we give both the scripts we have written so far to the same sprite, it will cycle through its costumes while it moves in a circle. Try it out if you like!

Interactive animation

We now have two different animated sprites, each doing their own thing. One of the special features of Scratch is that it makes *interactive* animation simple – we can program a sprite to react to you!

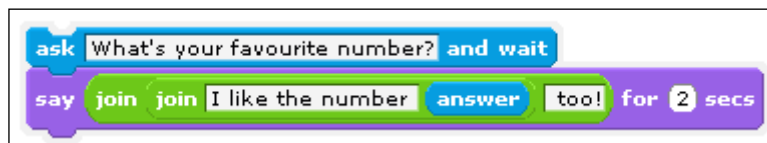
Create a new sprite in the same way as you did before using the **Choose new sprite from file** button. For this animation, we're going to have a simple conversation with the user, so choose an image of something that can talk. Find a good position and size for the new sprite, then build the following script in its script area:



There are a few new code blocks here. Let's go through them one by one:

- **ask and wait:** This gives the sprite a speech bubble, asking a question in a box. This block can be found in the **Sensing** section. The script will stop until the user has typed an answer into a text box on screen.
- **answer:** The answer to the question is stored in this block, and is also in the **Sensing** section.
- **join:** This takes two pieces of text and merges them into a longer sentence.
- **say:** This behaves a bit like **ask** did — the sprite uses a speech bubble to convey the text that is within its box. The **say** option can be found in the **Looks** section; there is also **think** which behaves similarly, but it has a thought bubble instead of a speech bubble. The number shown at the end of the **say** block determines how many seconds the speech bubble should be shown for.
- **stop all:** This ends all the scripts of all the sprites. This is not a necessary step, but is useful to end the scripts which run **forever**, and show that the animation has finished. This block can be found in the **Control** section.

Now let's continue the conversation. Add the following blocks between **say** and **stop all**, which you already have:



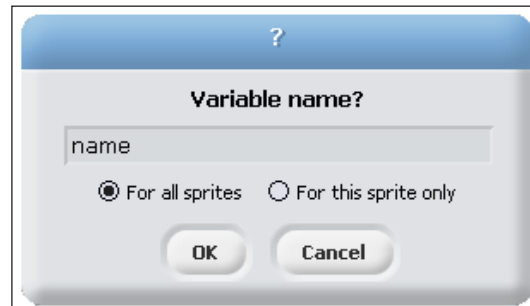
This is fairly similar to the previous section of code. The only clever part is that we want to join three pieces of text, but a single **join** block only allows two. To get around this, we join the first two pieces of text, **I like the number** and **answer**, in one block, and then join the whole first join block with **too!**.

Variables

Now, what if we want to use the player's name again to say **goodbye** at the end? We previously accessed the name using the **answer** block after we asked for the player's name. However, we have since asked another question, so **answer** now holds the player's favorite number.

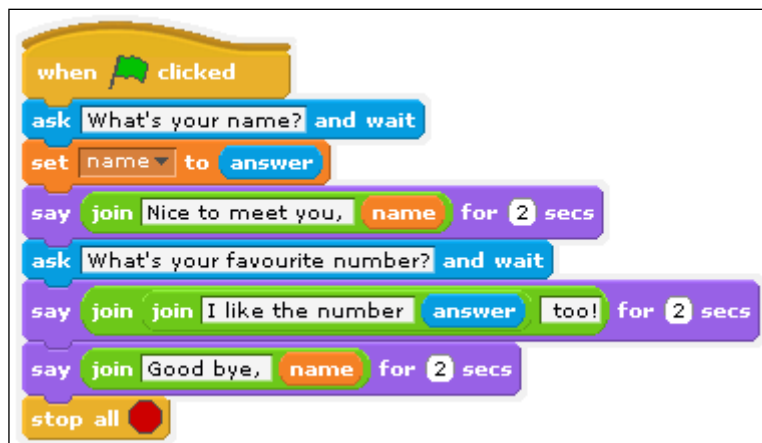
The solution to this is to use **variables**. A variable is used to store a piece of information, and this information can be changed at any time. The **answer** variable is a special variable which is automatically set whenever a question is asked.

1. Click on **Variables** and then on **Make a variable**. The following window will pop-up:



2. Call your variable *name* and click on **OK**. You will see that a few new block options have appeared at the left-hand side of the screen. You will also see that **name** appears with a tick next to it – this means that the current value of the name is being displayed on the stage. We don't need this to be done, so click on the tick to remove it.

We can now complete our short conversation by storing the player's name in the **name** variable, so that we can access it again when it's time to say goodbye. Here is the script used for the whole conversation:



As you can see, instead of using the **answer** block directly, we store its value in the **name** variable, and then use **name** throughout the script.

Movement

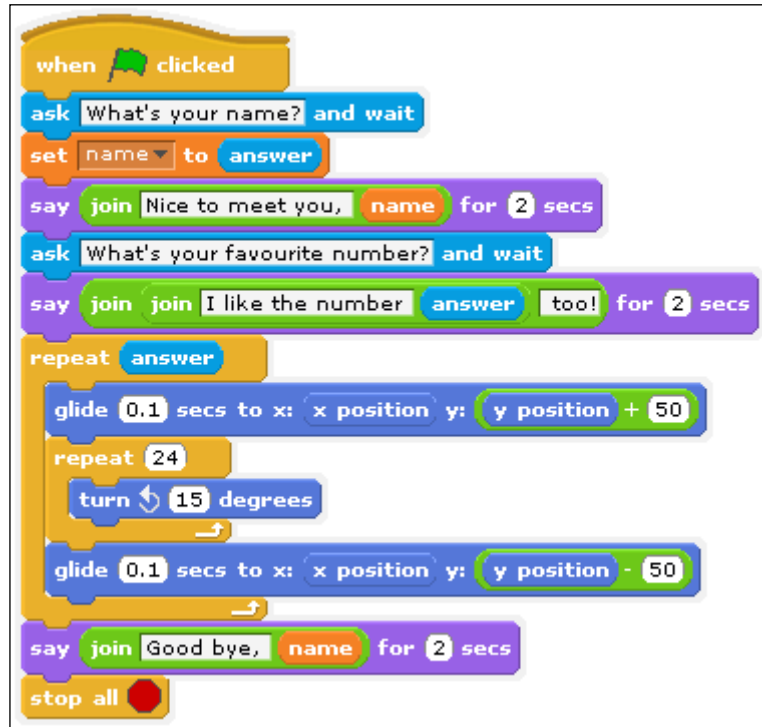
Now let's add a bit of movement to this animation. We're going to make the sprite do backflips, and the number of backflips will be decided by the player's favorite number. Here's the code needed to perform a single backflip:



You may find it useful to build this script, and then click on it to run it once, before adding it to the existing script. This is a good way to test that it works properly. Here's a quick explanation of what's happening in this code:

- **glide:** This is a smooth form of motion. We give it a duration and a position to move to. A shorter duration means quicker movement. The position is described using **x** and **y** coordinates. The value of **x** axis increases as you move to the right on the stage, and **y** increases as you move up the stage. You can see the current coordinates of the mouse cursor at the bottom-right of the stage. In this case, we want to move upwards at the start of the flip and downwards at the end, so we add 50 to the sprite's current **y position** at the start, and subtract the same 50 when we're finished.
- **repeat:** This makes all of the code blocks inside it run a certain number of times. It is a lot like **forever**, which we have already seen, but it will eventually stop. In this case, we make 24 small rotations to make the flip appear smooth. You might have noticed that 24 rotations of 15 degrees makes up the full 360 degrees of a circle, so the sprite will finish the right way up at the end.

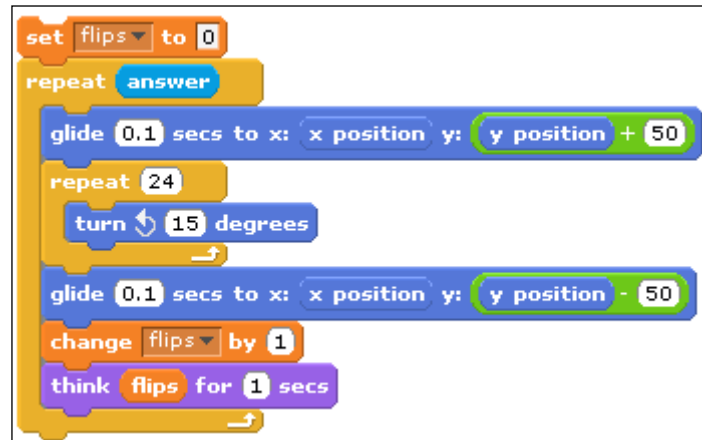
Now, that was one flip. We want to do a certain number of flips, so we will want to put the whole script, shown in the preceding image, inside another **repeat** block, and repeat it as many times as the player tells it to. In order to have access to the player's favorite number, we'll need to insert our new code into our previous script at the right point: after the question has been asked, but before saying goodbye. Here's what the code should look like now:



Keeping count

Now, let's have the sprite count the flips as it's moving so we know it does the right amount. For this, we're going to need a new variable to keep track of how many flips have taken place so far. Create a new variable called *flips* in the same way as you created *name* earlier. We will want to set the number of flips to zero before the sprite starts flipping; we will want to increase the number of flips by one after each flip, and also we'll also want the sprite to tell us how many flips it has done so far.

The following image shows you what the code for flipping should look like. All the other code should still be present in your script, but I'm not showing it in this image, so as to help you focus on the section that has changed:



As you can see, there are three new code blocks, which match up to the following three things we wanted to do:

- We **set flips to 0** at the start
- We **change flips by 1** after each flip (this adds 1 to the current value of *flips*)
- The sprite displays how many flips it has done so far in one second

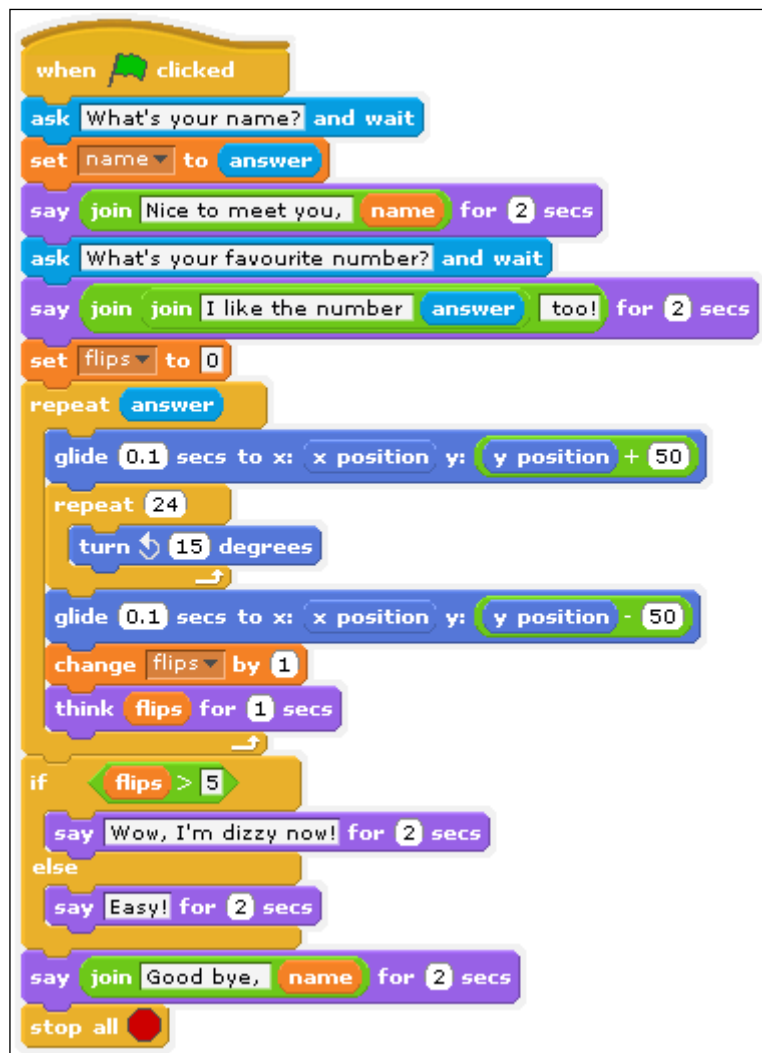
If-then-else

Repetition is one way to control which code blocks run; the **If-then-else** method is another. In the **Control** section, you will see a few different types of **repeat** and **forever** blocks, and also a couple of blocks that say **if**. These blocks allow us to *optionally* run any code blocks inside them, depending on the result of a *test*. The **else** version of the block gives us options: if the test passes, only the blocks in the first gap run, and if the test fails, only the blocks in the second gap run. We're going to use this block in our script.



Here, you are shown some code, which makes use of the **if-then-else** code block. Our test is performed to figure out whether the number of flips is more than five. If it is, the sprite tells us how dizzy it is. If it isn't, then the sprite says that it was easy to do all the flips. We can put as many code blocks as we like within each of the gaps in the **if-then-else** block but, for this example, we only have one block in each.

This code needs to go in after all the flips have finished, but before saying goodbye. Here's an illustration of the final, completed script:



Summary

In this chapter, we explored a few different ways of creating animations in Scratch. Along the way, we used a wide selection of the available code blocks. Many of these blocks are very similar to those used in other programming languages – you will recognize them in later chapters.

Importantly, you should recognize that this is your program, and you are free to change it as you like. You can change the way any of the sprites look, you can change any of the text or numbers, and you can even change which blocks are used to change the behavior of the program.

In the next chapter, we will continue the theme of making things interactive by building an entire game in Scratch – a version of *Angry Birds* which you will be able to modify to suit you.

3

Making Your Own Angry Birds Game

In this chapter, we are going to make our own version of the popular **Angry Birds™** game. What's more, when we're finished, we will be able to add all sorts of new rules and enemies to keep the game fresh. The following screenshot shows a completed version of our game:



If you haven't played Angry Birds before, here's a quick description of how the game works. The player launches a bird through the air using a slingshot and attempts to hit all the pigs at the other end of the level. In order to make things more challenging, the pigs are often hidden behind hills or inside flimsy buildings that the player must knock down.

By creating our own version of the game, we have the freedom to change whatever we like. We can change the level design, decrease gravity, fire the bird faster (or bee, in our case), replace all the characters, and add new power-ups and prizes. The sky is the limit!

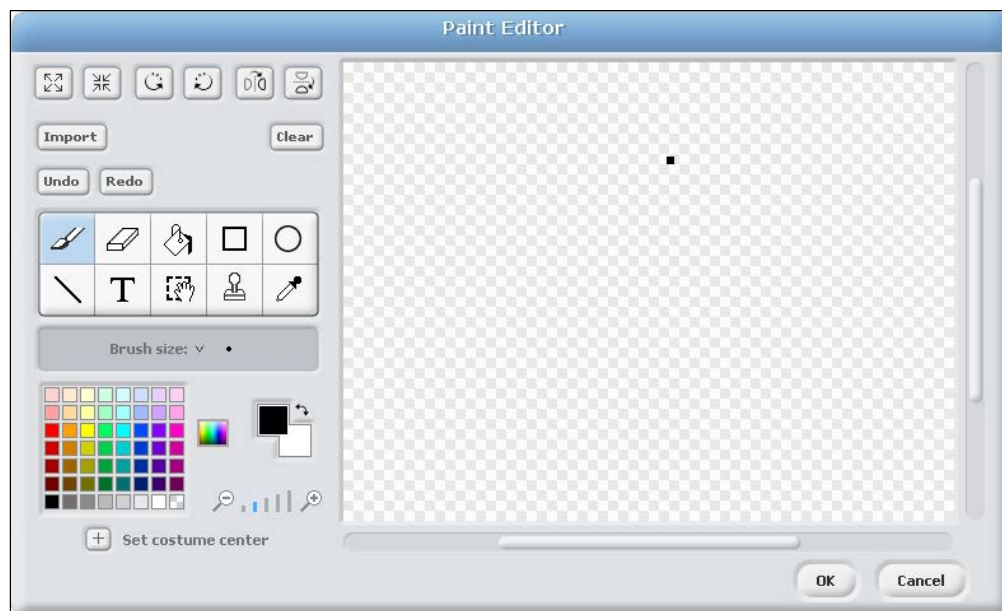
Creating a character

We're going to use the Scratch programming language to create this game. To start our game, we will need a character to fling through the air. Angry Birds, of course, uses birds as its main characters, but we can use whatever we like.

At the top of the sprite list you should see the three buttons, as shown in the following screenshot. The first lets you draw your own character, the second lets you use an existing image (including a wide range of images included in Scratch), and the third gives you a random image from Scratch's selection. We've only used the second button in the previous chapter, but now is a good chance to explore the others:



If you click on the first button, you will be shown the following window; it has plenty of easy-to-use options that can help you create your own drawings. Hover your mouse cursor over any of the buttons to see what they do:



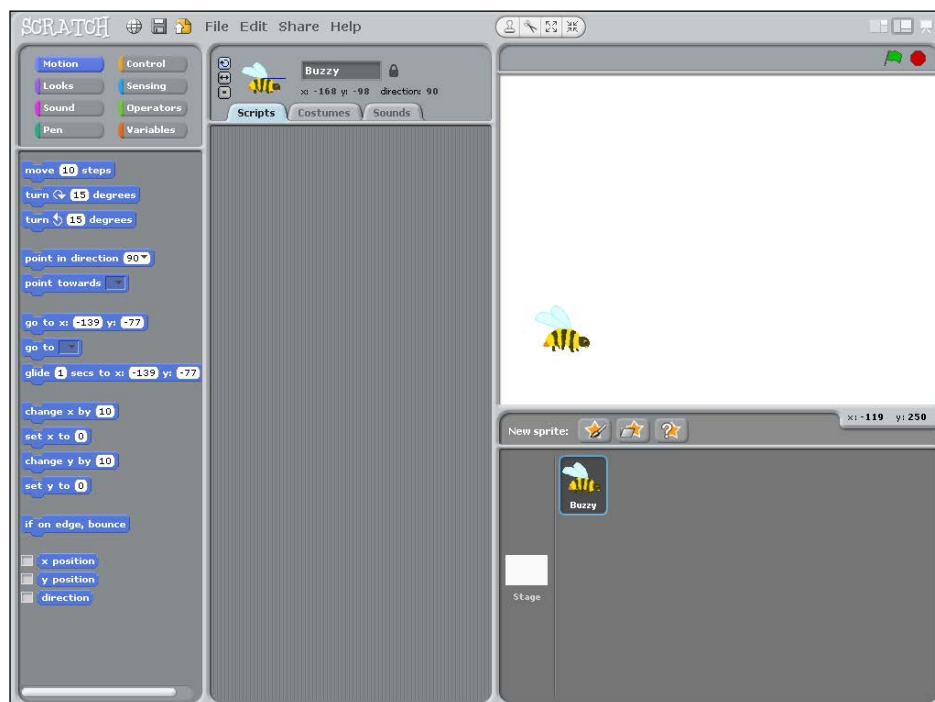
The second button brings up a fairly standard file explorer, which contains lots of neatly categorized images. This is the option I will use, but feel free to use something different.

Once you have drawn or selected a sprite, click on **OK** to add it to the game. If you choose not to use the default cat character, right-click on it in the sprite list and click on **Delete** (this will also delete any code you have created for the cat). You can navigate to **Edit | Undelete** to bring the cat and its code back.

The third button selects a random sprite and places it at a random position on the Stage.

Now that you have a main character, drag it within the Stage to roughly where you think will be a good starting position for it, and resize it by clicking on the **shrink** button in the sprite controls, and then repeatedly clicking on the sprite. I suggest making the sprite quite small so that there is plenty of room around it to fly. Now would also be a good time to give your character a name — there is a textbox at the top of the script area that should be named something similar to `Sprite 2`, which you can change to whatever you like.

Your screen should now look similar to this but with your own character instead of the bee that I have used:



Creating a level

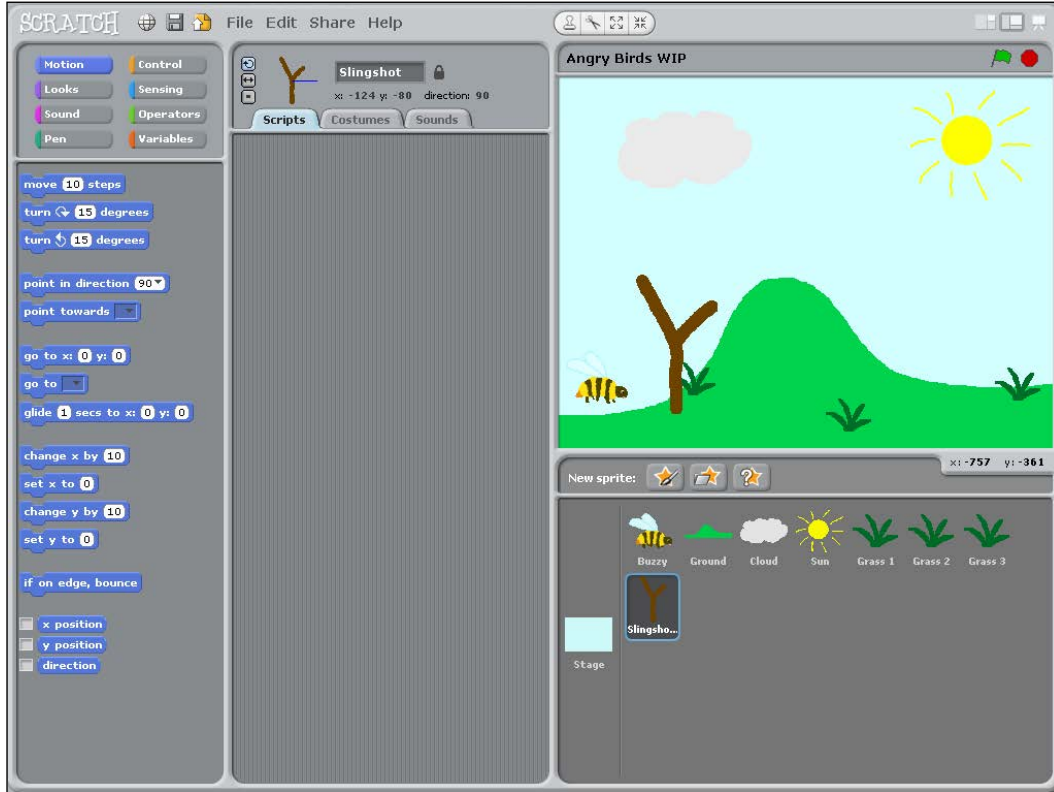
Now, let's make the game look a little more interesting by adding some scenery through following these steps:

1. To the left of the sprite list, you'll see a white rectangle called **Stage**. Click on it and then select the **Backgrounds** tab in the script area. Again, you have the option of drawing your own background or using a preexisting image, but this time, I recommend creating your own so that you can make the level fun to play.
2. Click on the **Edit** button. Try to keep your background as simple as possible; it will be easier to add extra objects (for example, the ground, trees, and clouds) as additional sprites later, because then you will be able to move them around more easily. It is perhaps easiest to simply fill the background with a solid sky blue color (and maybe some distant mountains).
3. Now back in the **Sprite** list, create sprites for all the scenery you want in your game. At the minimum, you will need to create a sprite for the ground, but you can add all sorts of little details. With each sprite you create, remember to position it on the **Stage**, make sure it is the size you want, and give it a descriptive name. Remember that you can duplicate sprites using the left button in the **Sprite Control** area. When you have finished, you might be left with something similar to the this screenshot:



I have put a hill in the middle of the level so as to make it more challenging to hit enemies on the right-hand side of the screen, when we create them.

When you are happy with your level design, draw a picture of a slingshot and add it to the left-hand side of the Stage. Give it the name `Slingshot` so we are able to find it easily later on. Your Scratch window should now look similar to this:



Moving the character

Now, let's start adding some code and making the game interactive! In this section, we'll do everything necessary to launch our main character using the slingshot.

Initialization

The first thing we want to do is make sure that the position of our main character resets every time we start the game. Click on the main character and create the following script in the script area:



The code snippet states that when the green flag is clicked, the current sprite (the main character) will move to the same position as the slingshot.

Check whether your code works by clicking on the green flag. You should see your character jumping to the same position as the slingshot. You may find that the character is behind the slingshot; if you would prefer for it to be in front, simply click on it on the Stage and drag it a short distance. Interacting with any sprite in this way will put it on top of all the other sprites.

Moving the character with the keyboard

Now, let's allow the player to move the character around using the keyboard so that they can aim their shot. We are mostly going to be making use of the code block (from the **Sensing** section), as seen in the following screenshot, but with different keys:



Before you read any further in this book, take a minute to have a look around at the available code blocks. Can you find any useful blocks that we could combine with this block to move a sprite up, down, left, or right? This block is a strange shape; how can we connect it to the motion blocks?

There are actually a few different ways to do this, but in this book, we will use the following code block:



Hopefully, this looks fairly sensible to you. If the left arrow key is pressed, do *something*. This *something* may be a bit confusing, however, so here's a quick explanation.

As mentioned briefly in the previous chapter, the position of every sprite on the screen is given by two numbers (or coordinates). The x coordinate tells you how far left or right the sprite is, and the y coordinate tells you how far up or down the sprite is. The center of the Stage is at $(0, 0)$, that is, both the x and y coordinates are zero. The x coordinate increases from left to right and the y coordinate increases from bottom to top. You can see the current coordinates of any sprite underneath its name in the script area, and the coordinates of the mouse are shown just under the **Stage**.

Since we want to move left when the left arrow key is pressed, we have to change the x coordinate by a negative amount. In this case, it has the same effect as subtracting 5.

We will need one of these code blocks for each arrow key:

- The left arrow key should change the x coordinate by -5
- The right arrow key should change the x coordinate by 5
- The up arrow key should change the y coordinate by 5
- The down arrow key should change the y coordinate by -5

Finally, since we want the player to be able to press each button multiple times to continue adjusting their position, we need to put all these blocks inside one big **forever** block. The **forever** block should be connected to the bottom of the existing script so that the player can adjust the character's position after the position has been reset. This is how your code should now look:

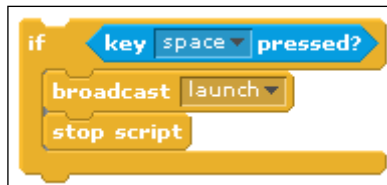


Once again, test your code out by clicking on the green flag. You should be able to move your character around by pressing the various arrow keys.

Launching the character!

Now that we've got the character in the right position, let's launch it! First, let's think about what we want to happen when the launch takes place. We want to stop the player from moving the character (so that they can't cheat), and instead, we want to start moving it with a speed and direction, depending on how far from the slingshot the player is.

Since we are moving into a new phase of the game, it is a good idea to use a separate script when we launch. This will help keep each script relatively small and manageable. Add the following code to the **forever** block where all of your other keyboard-handling blocks are, as follows:

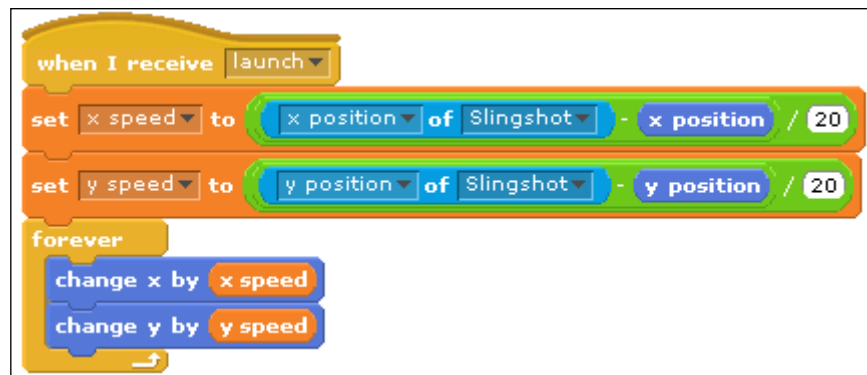


Here, **launch** is the name of a message. When the *Space* key is pressed, launch is sent to all the other scripts, and if any of them are waiting for this particular message, they will start to run. We also stop the current script so that we can stop repeatedly checking which keys are being pressed, and then the player can't continue to move the character around.

Before we create the second script, we want to be able to calculate how fast to fling the character. To do this, we are going to store the speed in a variable. Variables allow us to store one value each and can be shared between different scripts. In this case, we will store a number, but variables can also store text. We are actually going to use two variables to store the speed: one for up-down speed and the other for left-right speed. The reasons for this will become clear later. Create two new variables called *x speed* and *y speed*. Make sure they are both valid for this sprite only by choosing the appropriate option when creating the variables.

Flight

Now that we have these variables, we can create the second script, which will control our flight through the air. The code for this is shown in the following screenshot. It's a little complicated, but try to work out what it does as you build the script up in the script area. I'll explain how it works shortly (for the two long blocks that are almost identical, it is possible to create one, then right-click on it and duplicate it to save effort in creating the second one):



This script waits until it receives the **launch** message from the first script. Only then does it start. We set the **x speed** variable to a value that is relative to the distance between the character and the slingshot; this distance is calculated by subtracting the position of the bee from the position of the slingshot. I divide the value by **20** to make sure that the flight isn't too fast, but you may prefer a different value here. Here are step-by-step instructions to build the code block up:

- Start with a **set x speed** block
- Insert a division block and type **20** into the second space
- Insert a subtraction block into the first space of the division block
- Insert **x position of Slingshot** into the first space of the subtraction block
- Insert **x position** into the second space of the subtraction block

We then do exactly the same to compute the **y speed** value. Once the speed has been computed, we repeatedly move the object according to our speed.

We're now in a good place to test if everything is working. Click on the green flag, move around, and then launch using the *Spacebar*. You should be able to see your character fly in a straight line across the screen. You may want to try launching from different positions to see how this affects your speed and direction.

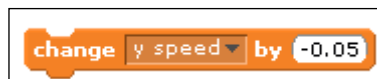
One thing that you may have noticed is that your character flies directly through the middle of the slingshot, not the part that it should actually be fired from. This is easy to fix. Click on **Slingshot** in the sprite list; choose the **Costumes** tab in the script area, click on **Edit**, and click on **Set costume center**. You can now drag the crosshairs around to choose a more sensible launch position. Once you have finished, click on **OK**. The slingshot will probably need repositioning on **Stage**, but your character's flight should now follow a better path.

Adding physics

The next thing for us to do is to give our character a more interesting flight path. The game would be too easy (and no fun) if we just flew in a straight line through all the obstacles.

Gravity

First, let's add some gravity. Gravity has the effect of pulling objects down toward the ground. How can we model gravity in our game? The answer lies in the way we split our speed into both **x speed** and **y speed**. Gravity will only affect **y speed**, our speed in the up-down direction, so we can leave **x speed** as it is. Since the *y* coordinate increases as we move up but gravity pulls us down, we want gravity to keep subtracting a small amount from **y speed**. Add the following code block inside the **forever** block of your second script:



Try out the game now. You should arc through the air until you hit one of the edges of the screen. You may tweak the number in this code block if you wish; a higher negative number will give you stronger gravity. What happens if the number is positive?

Bouncing

Next, we'll make something more interesting happen if we hit the edge of the screen. As always, there are several options available, but I am going to suggest bouncing off the edges. When we bounce, we want to have the same speed but travel in the opposite direction. When we hit either of the side edges, we want our left-right direction to change, and when we hit the top or bottom edges, we want our up-down direction to change. There is an **if on edge, bounce** block in the **Motion** section, but it can have some unexpected effects in the game. Add it inside the **forever** block to see the effects, if you like, but remember to remove it again before continuing.

Instead, we'll write our own code to handle bouncing. Add the following code inside the **forever** block:



All we're doing here is checking the current position to see if it is at an edge and then reversing the direction. The numbers **240** and **180** come from the width and height of the Stage, respectively, and multiplying by **-1** is a good way to keep the speed the same but reverse the direction.

Test the game again. Your character should bounce around the screen in smooth, curved paths.

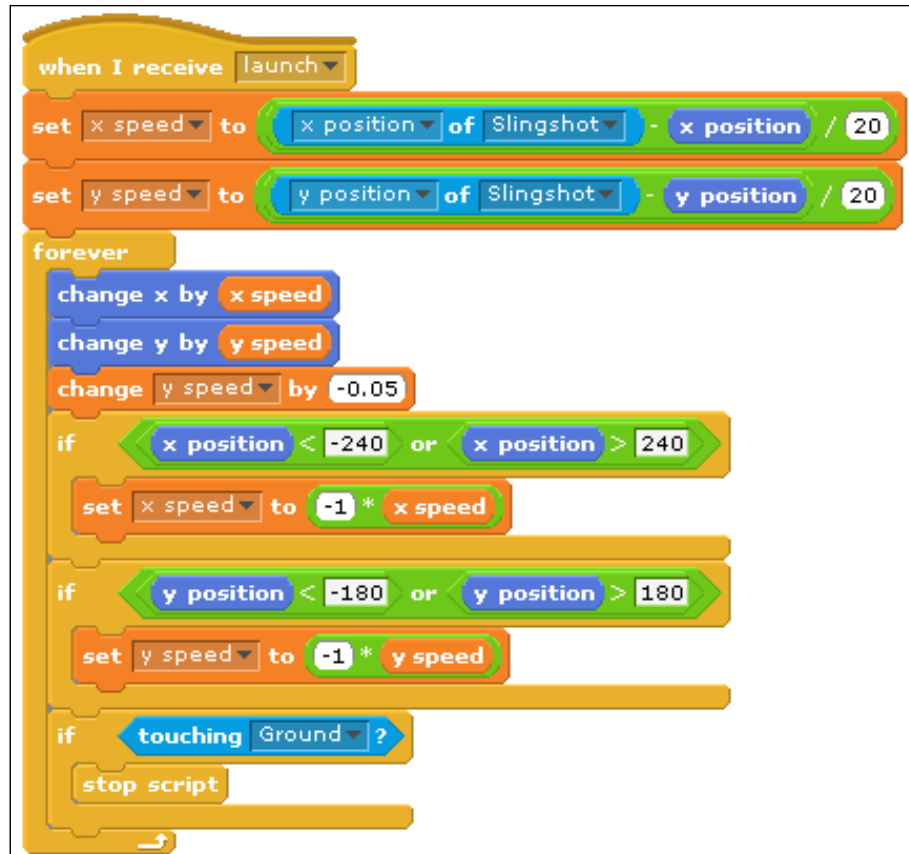
Ending the game

The problem now is that you bounce around forever. We want the bouncing to stop at some point, and a good time to do this is when the character hits the ground. This is easy to do in Scratch with the following code:



Add this inside the **forever** block, and the script will end when the character sprite hits the ground sprite (you will need to choose the name of the sprite that you used for the ground). Since this script is in control of the character's movement, ending the script ends the movement, which is what we wanted.

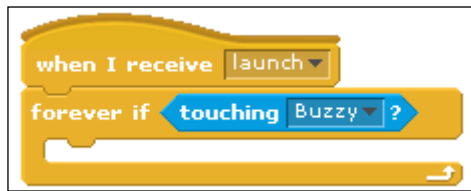
Give your physics of the game a final test by playing the game. Your character should fly through the air while being pulled downwards by gravity, bounce off the edges of the screen, and stop when it hits the ground. This is how your second script should now look:



Scoring

Now that our main character can be launched properly, it's time to give the player something to aim at. In Angry Birds, there are pigs, but we can have anything we like. Draw a new sprite or use an existing one in the same way we created the main character earlier. I am going to use a premade shark in this example. Resize the sprite and put it in a good position.

Do you remember how we checked to see when the main character hit the ground? We're going to need to do something very similar here to detect when an enemy is hit by the main character. The following is the main piece of code to detect collisions, and inside it, we're going to put all the effects we want to happen when the enemy is hit. Make sure the enemy sprite is selected when you create this script – it controls the enemy's behavior and not the main character's. Note that we're using **forever if** rather than just **if** as we want to keep checking for collisions. Buzzy is the name of the sprite for my main bee character:



If everything has been done correctly, when an enemy is hit, the following events occur:

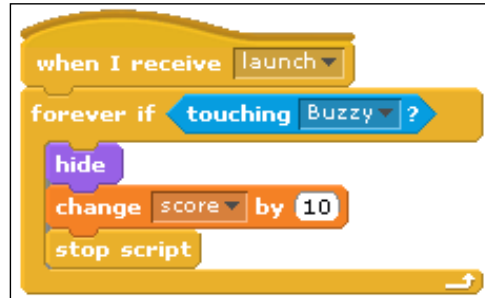
- The enemy disappears
- The score is updated
- The script for this enemy stops – we don't care about any future collisions

We can make the enemy disappear using the **hide** code block, and we've already seen how the script can be ended using **stop script**. The only thing left, then, is the score.

Create a new variable called `score`, and this time make sure that **For all sprites** is selected. This ensures that all the sprites have access to the score, so if there are multiple enemies, they can all update the same variable. Once the variable has been created, make sure the box next to it is marked so that the score appears on the Stage.

Now, we need to add some code so that the score increases when the enemy is hit. Add **change score by 10** inside the **forever if** block.

Your script should now look similar to this screenshot:



Test your game by playing it and try to hit this enemy. Remember, if the game is too hard or too easy, you can adjust the sizes of the sprites, their launch speed, and the gravity. You will notice that once you hit the enemy, it disappears and you get points, but the score and enemies don't reset when you play again. Let's fix this. Add the code shown in the following screenshot to the enemy sprite as a second script:



Now the enemy should come back, and the score should reset to 0 every time you click on the green flag.

Only when you've finished all the code should you create multiple enemies. Right-click on the enemy sprite in the sprite list and click on **Duplicate** to create a copy. This copy will have all the necessary code in it to update the score, and then disappear after it has been hit. Create as many copies as you like, place them wherever you like, and then sit back and enjoy your game! It should look similar to this screenshot:



Extensions

So far, we have created the bare minimum required for a game. There are all sorts of extra features we could add, such as the following:

1. An end-game screen, which shows when all the enemies have been hit or when the main character touches the ground.
2. Animation when two sprites collide.
3. A special enemy that gives bonus points.
4. Barriers that slow the player down.
5. Power-ups that increase the player's speed or flip gravity, for example.
6. Extra controls so that the player can continue to affect the character after it has been launched.

I will leave the rest of your game up to you, but here are some example scripts to give you some ideas. Try to work out what they do and where they might go, or just try them out! Some scripts might require minor modifications elsewhere to fit in properly:



Summary

In this chapter, we continued to learn how to use the Scratch programming language, and we went as far as creating an entire game.

In the next chapter, we'll take our knowledge of Scratch and see how we can apply it to a different programming language called Python. There, we'll learn how to use randomness to create lighthearted insults.

4

Creating Random Insults

In this chapter, we're going to move on from Scratch and use the **Python** programming language to generate random funny phrases such as, *Alice has a smelly foot!*

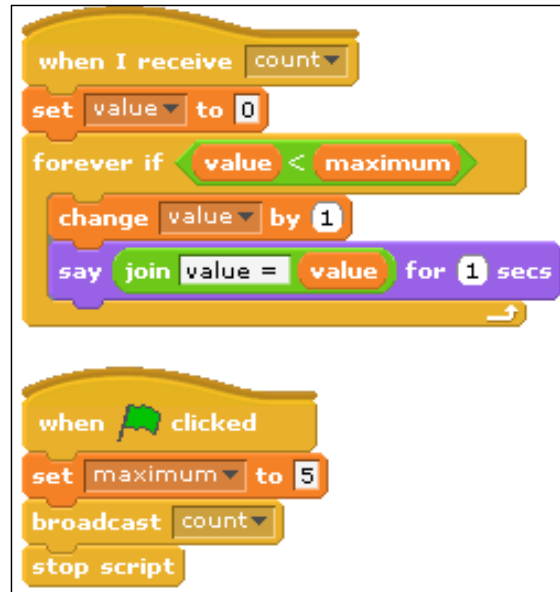
Python

In this chapter, we are going to use the Python programming language. Almost all programming languages are capable of doing the same things, but they are usually designed with different specializations. Some languages are designed to perform one job particularly well, some are designed to run code as fast as possible, and some are designed to be easy to learn.

Scratch was designed to develop animations and games, and to be easy to read and learn, but it can be difficult to manage large programs. Python is designed to be a good general-purpose language. It is easy to read and can run code much faster than Scratch.

Python is a text-based language. Using it, we type the code rather than arrange building blocks. This makes it easier to go back and change the pieces of code that we have already written, and it allows us to write complex pieces of code more quickly. It does mean that we need to type our programs accurately, though – there are no limits to what we can type, but not all text will form a valid program. Even a simple spelling mistake can result in errors. Lots of tutorials and information about the available features are provided online at <http://docs.python.org/2/>. *Learn Python the Hard Way*, by *Shaw Zed A.*, is another good learning resource, which is available at <http://learnpythonthehardway.org>.

As an example, let's take a look at some Scratch and Python code, respectively, both of which do the same thing. Here's the Scratch code:



The Python code that does the same job looks like:

```
def count(maximum):  
    value = 0  
    while value < maximum:  
        value = value + 1  
        print "value =", value  
  
count(5)
```

Even if you've never seen any Python code before, you might be able to read it and tell what it does. Both the Scratch and Python code count from 0 to a maximum value, and display the value each time.

The biggest difference is in the first line. Instead of waiting for a message, we **define** (or create) a **function**, and instead of sending a message, we **call** the function (more on how to run Python code, shortly). Notice that we include `maximum` as an **argument** to the `count` function. This tells Python the particular value we would like to keep as the maximum, so we can use the same code with different maximum values.

The other main differences are that we have `while` instead of `forever` `if`, and we have `print` instead of `say`. These are just different ways of writing the same thing. Also, instead of having a block of code wrap around other blocks, we simply put an extra four spaces at the beginning of a line to show which code is contained within a particular block.

Python programming

To run a piece of Python code, open **Python 2** from the **Programming** menu on the Raspberry Pi desktop and perform the following steps:

1. Type the previous code into the window and you should notice that it can recognize how many spaces to start a line with.
2. When you have finished the function block, press *Enter* a couple of times, until you see `>>>`. This shows that Python recognizes that your block of code has been completed, and that it is ready to receive a new command.
3. Now, you can run your code by typing in `count (5)` and pressing *Enter*. You can change 5 to any number you like and press *Enter* again to count to a different number.

We're now ready to create our program!



The Raspberry Pi also supports Python 3, which is very similar but incompatible with Python 2. This book uses Python 2 as it is the default, but you may like to explore the differences online at http://python-future.org/compatible_idioms.html.

The program we're going to use to generate phrases

As mentioned earlier, our program is going to generate random, possibly funny, phrases for us. To do this, we're going to give each phrase a common structure, and randomize the word that appears in each position. Each phrase will look like:

```
<name> has a <adjective> <noun>
```

Where `<name>` is replaced by a person's name, `<adjective>` is replaced by a descriptive word, and `<noun>` is replaced by the name of an object.

This program is going to be a little larger than our previous code example, so we're going to want to save it and modify it easily. Navigate to **File | New Window** in Python 2. A second window will appear which starts off completely blank. We will write our code in this window, and when we run it, the results will appear in the first window. For the rest of the chapter, I will call the first window the **Shell**, and the new window the **Code Editor**. Remember to save your code regularly!

Lists

We're going to use a few different **lists** in our program. Lists are an important part of Python, and allow us to group together similar things. In our program, we want to have separate lists for all the possible names, adjectives, and nouns that can be used in our sentences. We can create a list in this manner:

```
names = ["Alice", "Bob", "Carol"]
```

Here, we have created a variable called `names`, which is a list. The list holds three items or **elements**: `Alice`, `Bob`, and `Carol`. We know that it is a list because the elements are surrounded by square brackets, and are separated by commas. The names need to be in quote marks to show that they are text, and not the names of variables elsewhere in the program.

To access the elements in a list, we use the number which matches its position, but curiously, we start counting from zero. This is because if we know where the start of the list is stored, we know that its first element is stored at position `start + 0`, the second element is at position `start + 1`, and so on. So, `Alice` is at position 0 in the list, `Bob` is at position 1, and `Carol` is at position 2. We use the following code to display the first element (`Alice`) on the screen:

```
print names[0]
```

We've seen `print` before: it displays text on the screen. The rest of the code is the name of our list (`names`), and the position of the element in the list that we want surrounded by square brackets.


Type these two lines of code into the Code Editor, and then navigate to **Run | Run Module** (or press `F5`). You should see **Alice** appear in the Shell. Feel free to play around with the names in the list or the position that is being accessed until you are comfortable with how lists work. You will need to rerun the code after each change. What happens if you choose a position that doesn't match any element in the list, such as 10?

Adding randomness

So far, we have complete control over which name is displayed. Let's now work on displaying a random name each time we run the program. Update your code in the Code Editor so it looks like:

```
import random
names = ["Alice", "Bob", "Carol"]
position = random.randrange(3)
print names[position]
```

In the first line of the code, we import the `random` module. Python comes with a huge amount of code that other people have written for us, separated into different **modules**. Some of this code is simple, but makes life more convenient for us, and some of it is complex, allowing us to reuse other people's solutions for the challenges we face and concentrate on exactly what we want to do. In this case, we are making use of a collection of functions that deal with random behavior. We must import a module before we are able to access its contents.

[ Information on the available modules available can be found online at www.python.org/doc/.]

After we've created the list of names, we then compute a random position in the list to access. The name `random.randrange` tells us that we are using a function called `randrange`, which can be found inside the `random` module that we imported earlier. The `randrange` function gives us a random whole number less than the number we provide. In this case, we provide 3 because the list has three elements and we store the random position in a new variable called `position`. Finally, instead of accessing a fixed element in the `names` list, we access the element that `position` refers to.

If you run this code a few times, you should notice that different names are chosen randomly.

Now, what happens if we want to add a fourth name, `Dave`, to our list? We need to update the list itself, but we also need to update the value we provide to `randrange` to let it know that it can give us larger numbers. Making multiple changes just to add one name can cause problems—if the program is much larger, we may forget which parts of the code need to be updated. Luckily, Python has a nice feature which allows us to make this simpler.

Instead of a fixed number (such as 3), we can ask Python for the length of a list, and provide that to the `randrange` function. Then, whenever we update the list, Python knows exactly how long it is, and can generate suitable random numbers. Here is the code, which is updated to make it easier to change the length of the list:

```
import random
names = ["Alice", "Bob", "Carol"]
length = len(names)
position = random.randrange(length)
print names[position]
```

Here, we've created a new variable called `length` to hold the length of the list. We then use the `len` function (which is short for *length*) to compute the length of our list, and we give `length` to the `randrange` function. If you run this code, you should see that it works exactly as it did before, and it easily copes if you add or remove elements from the list.

It turns out that this is such a common thing to do, that the writers of the `random` module have provided a function which does the same job. We can use this to simplify our code:

```
import random
names = ["Alice", "Bob", "Carol", "Dave"]
print random.choice(names)
```

As you can see, we no longer need to compute the length of the list or a random position in it: `random.choice` does all of this for us, and simply gives us a random element of any list we provide it with. As we will see in the next section, this is useful since we can reuse `random.choice` for all the different lists we want to include in our program.

If you run this program, you will see that it works the same as it did before, despite being much shorter.

Creating phrases

Now that we can get a random element from a list, we've crossed the halfway mark to generating random sentences!

Create two more lists in your program, one called `adjectives`, and the other called `nouns`. Put as many descriptive words as you like into the first one, and a selection of objects into the second. Here are the three lists I now have in my program:

```
names = ["Alice", "Bob", "Carol", "Dave"]
adjectives = ["fast", "slow", "pretty", "smelly"]
nouns = ["dog", "car", "face", "foot"]
```

Also, instead of printing our random elements immediately, let's store them in variables so that we can put them all together at the end. Remove the existing line of code with `print` in it, and add the following three lines after the lists have been created:

```
name = random.choice(names)
adjective = random.choice(adjectives)
noun = random.choice(nouns)
```

Now, we just need to put everything together to create a sentence. Add this line of code right at the end of the program:

```
print name, "has a", adjective, noun
```

Here, we've used commas to separate all of the things we want to display. The `name`, `adjective`, and `noun` are our variables holding the random elements of each of the lists, and `"has a"` is some extra text that completes the sentence. `print` will automatically put a space between each thing it displays (and start a new line at the end). If you ever want to prevent Python from adding a space between two items, separate them with `+` rather than a comma.

That's it! If you run the program, you should see random phrases being displayed each time, such as *Alice has a smelly foot* or *Carol has a fast car*.

Making mischief

So, we have random phrases being displayed, but what if we now want to make them *less* random? What if you want to show your program to a friend, but make sure that it only ever says nice things about you, or bad things about them? In this section, we'll extend the program to do just that.

Dictionaries

The first thing we're going to do is replace one of our lists with a **dictionary**. A dictionary in Python uses one piece of information (a number, some text, or almost anything else) to search for another. This is a lot like the dictionaries you might be used to, where you use a word to search for its meaning. In Python, we say that we use a **key** to look for a **value**.

We're going to turn our `adjectives` list into a dictionary. The keys will be the existing descriptive words, and the values will be tags that tell us what sort of descriptive words they are. Each adjective will be `"good"` or `"bad"`.

My `adjectives` list becomes the following dictionary. Make similar changes to yours.

```
adjectives = {"fast": "good", "slow": "bad", "pretty": "good",
             "smelly": "bad"}
```

As you can see, the square brackets from the list become curly braces when you create a dictionary. The elements are still separated by commas, but now each element is a **key-value pair** with the adjective first, then a colon, and then the type of adjective it is.

To access a value in a dictionary, we no longer use the number which matches its position. Instead, we use the key with which it is paired. So, as an example, the following code will display "good" because "pretty" is paired with "good" in the `adjectives` dictionary:

```
print adjectives["pretty"]
```

If you try to run your program now, you'll get an error which mentions `random.choice(adjectives)`. This is because `random.choice` expects to be given a list, but is now being given a dictionary. To get the code working as it was before, replace that line of code with this:

```
adjective = random.choice(adjectives.keys())
```

The addition of `.keys()` means that we only look at the keys in the dictionary — these are the adjectives we were using before, so the code should work as it did previously. Test it out now to make sure.

Loops

You may remember the `forever` and `repeat` code blocks in Scratch. In this section, we're going to use Python's versions of these to repeatedly choose random items from our dictionary until we find one which is tagged as "good". A **loop** is the general programming term for this repetition — if you walk around a loop, you will repeat the same path over and over again, and it is the same with loops in programming languages.

Here is some code, which finds an adjective and is tagged as "good". Replace your existing `adjective =` line of code with these lines:

```
while True:
    adjective = random.choice(adjectives.keys())
    if adjectives[adjective] == "good":
        break
```

The first line creates our loop. It contains the `while` key word, and a test to see whether the code should be executed inside the loop. In this case, we make the test `True`, so it always passes, and we always execute the code inside. We end the line with a colon to show that this is the beginning of a block of code. While in Scratch we could drag code blocks inside of the `forever` or `repeat` blocks, in Python we need to show which code is inside the block in a different way. First, we put a colon at the end of the line, and then we indent any code which we want to repeat by four spaces.

The second line is the code we had before: we choose a random adjective from our dictionary.

The third line uses `adjectives[adjective]` to look into the (`adjectives`) dictionary for the tag of our chosen adjective. We compare the tag with `"good"` using the `double = sign` (a `double =` is needed to make the comparison different from the `single =` case, which stores a value in a variable). Finally, `if` the tag matches `"good"`, we enter another block of code: we put a colon at the end of the line, and the following code is indented by another four spaces. This behaves the same way as the Scratch `if` block.

The fourth line contains a single word: `break`. This is used to escape from loops, which is what we want to do now that we have found a `"good"` adjective.

If you run your code a few times now, you should see that none of the bad adjectives ever appear.

Conditionals

In the preceding section, we saw a simple use of the `if` statement to control when some code was executed. Now, we're going to do something a little more complex. Let's say we want to give Alice a good adjective, but give Bob a bad adjective. For everyone else, we don't mind if their adjective is good or bad.

The code we already have to choose an adjective is perfect for Alice: we always want a good adjective. We just need to make sure that it only runs if our random phrase generator has chosen Alice as its random person. To do this, we need to put all the code for choosing an adjective within another `if` statement, as shown here:

```
if name == "Alice":
    while True:
        adjective = random.choice(adjectives.keys())
        if adjectives[adjective] == "good":
            break
```

Remember to indent everything inside the `if` statement by an extra four spaces.

Next, we want a very similar piece of code for Bob, but also want to make sure that the adjective is bad:

```
elif name == "Bob":
    while True:
        adjective = random.choice(adjectives.keys())
        if adjectives[adjective] == "bad":
            break
```

The only differences between this and Alice's code is that the name has changed to "Bob", the target tag has changed to "bad", and `if` has changed to `elif`. The word `elif` in the code is short for *else if*. We use this version because we only want to do this test if the first test (with Alice) fails. This makes a lot of sense if we look at the code as a whole: *if* our random person is Alice, do something, *else if* our random person is Bob, do something else.

Finally, we want some code that can deal with everyone else. This time, we don't want to perform another test, so we don't need an `if` statement: we can just use `else`:

```
else:
    adjective = random.choice(adjectives.keys())
```

With this, our program does everything we wanted it to do. It generates random phrases, and we can even customize what sort of phrase each person gets. You can add as many extra `elif` blocks to your program as you like, so as to customize it for different people.

Functions

In this section, we're not going to change the behavior of our program at all; we're just going to tidy it up a bit.

You may have noticed that when customizing the types of adjectives for different people, you created multiple sections of code, which were almost identical. This isn't a very good way of programming because if we ever want to change the way we choose adjectives, we will have to do it multiple times, and this makes it much easier to make mistakes or forget to make a change somewhere.

What we want is a single piece of code, which does the job we want it to do, and then be able to use it multiple times. We call this piece of code a **function**. We saw an example of a function being created in the comparison with Scratch at the beginning of this chapter, and we've used a few functions from the `random` module already. A function can take some inputs (called **arguments**) and does some computation with them to produce a result, which it **returns**.

Here is a function which chooses an adjective for us with a given tag:

```
def chooseAdjective(tag):  
    while True:  
        item = random.choice(adjectives.keys())  
        if adjectives[item] == tag:  
            break  
    return item
```

In the first line, we use `def` to say that we are **defining** a new function. We also give the function's name and the names of its arguments in brackets. We separate the arguments by commas if there is more than one of them. At the end of the line, we have a colon to show that we are entering a new code block, and the rest of the code in the function is indented by four spaces.

The next four lines should look very familiar to you – they are almost identical to the code we had before. The only difference is that instead of comparing with "good" or "bad", we compare with the `tag` argument. When we use this function, we will set `tag` to an appropriate value.

The final line returns the suitable adjective we've found. Pay attention to its indentation. The line of code is inside the function, but not inside the `while` loop (we don't want to return every item we check), so it is only indented by four spaces in total.

Type the code for this function anywhere above the existing code, which chooses the adjective; the function needs to exist in the code prior to the place where we use it. In particular, in Python, we tend to place our code in the following order:

1. Imports
2. Functions
3. Variables
4. Rest of the code

This allows us to use our functions when creating the variables. So, place your function just after the `import` statement, but before the lists. We can now use this function instead of the several lines of code that we were using before. The code I'm going to use to choose the adjective now becomes:

```
if name == "Alice":  
    adjective = chooseAdjective("good")  
elif name == "Bob":  
    adjective = chooseAdjective("bad")  
else:  
    adjective = random.choice(adjectives.keys())
```

This looks much neater! Now, if we ever want to change how an adjective is chosen, we just need to change the `chooseAdjective` function, and the change will be seen in every part of the code where the function is used.

Complete code listing

Here is the final code you should have when you have completed this chapter. You can use this code listing to check that you have everything in the right order, or look for other problems in your code. Of course, you are free to change the contents of the lists and dictionaries to whatever you like; this is only an example:

```
import random

def chooseAdjective(tag):
    while True:
        item = random.choice(adjectives.keys())
        if adjectives[item] == tag:
            break
    return item

names = ["Alice", "Bob", "Carol", "Dave"]
adjectives = {"fast":"good", "slow":"bad", "pretty":"good",
             "smelly":"bad"}
nouns = ["dog", "car", "face", "foot"]

name = random.choice(names)
#adjective = random.choice(adjectives)
noun = random.choice(nouns)

if name == "Alice":
    adjective = chooseAdjective("good")
elif name == "Bob":
    adjective = chooseAdjective("bad")
else:
    adjective = random.choice(adjectives.keys())

print name, "has a", adjective, noun
```

Summary

In this chapter, we learned about the Python programming language and how it can be used to create random phrases. We saw that it shared lots of features with Scratch, but is simply presented differently.

In the next chapter, we'll continue learning about Python, and will discover how to allow computer code to interact with the physical world.

5

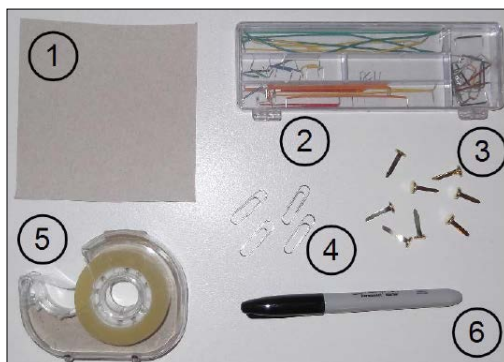
Testing Your Speed

In this chapter, we're going to create a new game that will test how quickly the player can react. To do this, we will create our own game controller—something you can't easily do on a normal computer—and write a program that tells the Pi what to do when the controller's buttons are pressed.

If you do not have the components required to create the controller, an alternative program that uses the keyboard instead is provided at the end of the chapter. It is very similar to the default program, so it is still worth reading through this chapter to learn how everything works.

Materials needed to make your own controller

The materials that are required to make your own controller are shown in the following figure. Think about what you would like your controller to look like and how many buttons it should have, as this will determine how many of each item you will need:



The preceding figure shows the items we need to create a controller and the details are as follows:

- Card (1) (as large as you want the controller to be)
- Wires (2)
- Paper fasteners (3) (2 x number of buttons)
- Paper clips (4) (1 x number of buttons; each clip should be in plain metal with no coating around it)
- Sticky tape (5)
- Pens/pencils for decoration (6)

If you already have some electric switches you'd like to use, you will not need the paper fasteners, paper clips, or sticky tape, but I think it's more fun to make everything from scratch!

We also need a safe way to connect the wires to the Raspberry Pi. One approach is to use special male to female wires, which behave like normal wires at one end, but can be connected to a pin or another wire at the other end. The other way is to use a Raspberry Pi **breakout board**, ribbon cable, and a breadboard. These three work together to give a larger area to plug in electronic components. The website <http://www.adafruit.com> is a great online shop that sells these sorts of components for the Raspberry Pi; you even get explanations on how to use them (refer to <http://www.adafruit.com/category/105>).

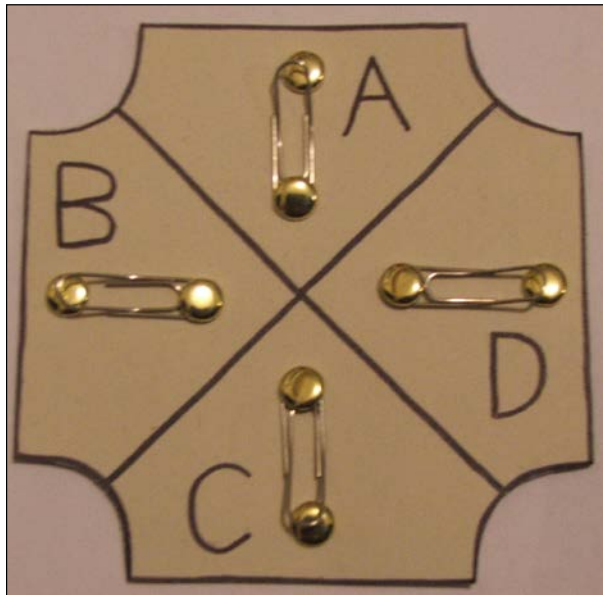
Creating the game controller

In order to design a controller, we first need to know what sort of game is going to be played. I am going to explain how to make a game where the player is given a letter and they have to press the button of that letter as quickly as possible. They are then told another letter. The player has to hit as many buttons correctly as they can in a 30-second time limit.

There are many ways in which this game can be varied; instead of ordering the player to press a particular button, the game could ask the player a multiple-choice question, and instead of letters, the buttons could be labeled with Yes, No, Maybe, or different colors. You could give the player multiple commands in a sequence and make sure that they press all the buttons in the right order. It would even be possible to make a huge controller and treat it as more of a board game. I will leave the game design up to you, but I recommend that you follow the instructions in this chapter until the end and then change things to your liking once you know how everything works.

The controller base

So, now that we know how the game is going to be played, it's time to design the controller. This is what my finished controller design looks like, with four different letters and its paper clip buttons in place:



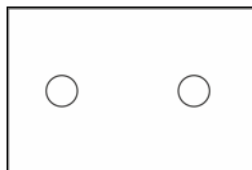
Make sure each button area is at least a little bigger than a paper clip, as these are what the buttons will be made of. I recommend a maximum of eight buttons.

Draw your design on to the card, decorate it however you like, and then cut it out.

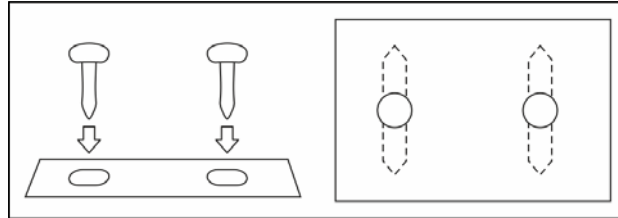
Adding buttons

Now for each button, we need to perform the following steps:

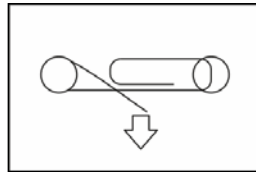
1. Poke two small holes in the card, roughly 3 cm apart (or however long your paper clips are), as shown in the following figure. Use a sharp pencil or a pair of scissors to do this:



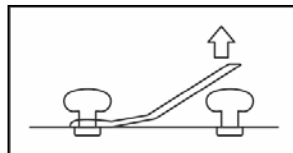
2. Push a paper fastener through each hole and open them out:



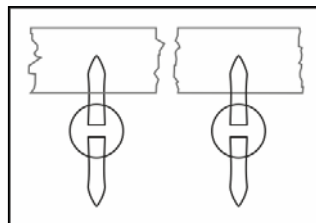
3. Wrap a paper clip around the head of one of the fasteners, and (if necessary) bend it so that it grips the fastener tightly:



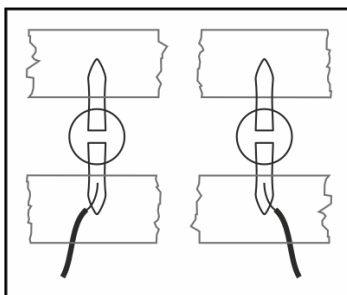
4. Bend the other end of the paper clip up very slightly, so that it doesn't touch the second fastener unless you press down on it:



5. Turn the card over and tape one leg of each fastener in place, making sure that they don't touch each other:



6. Tape a length of wire to each of the two remaining legs of the fasteners. The ends of the wires should be exposed metal so that electricity can flow through the wire, paper fastener, and paper clip (as shown in the following figure). You might want to delay this step until later, when you have a better idea of how long the wire should be:



Connecting to the Raspberry Pi

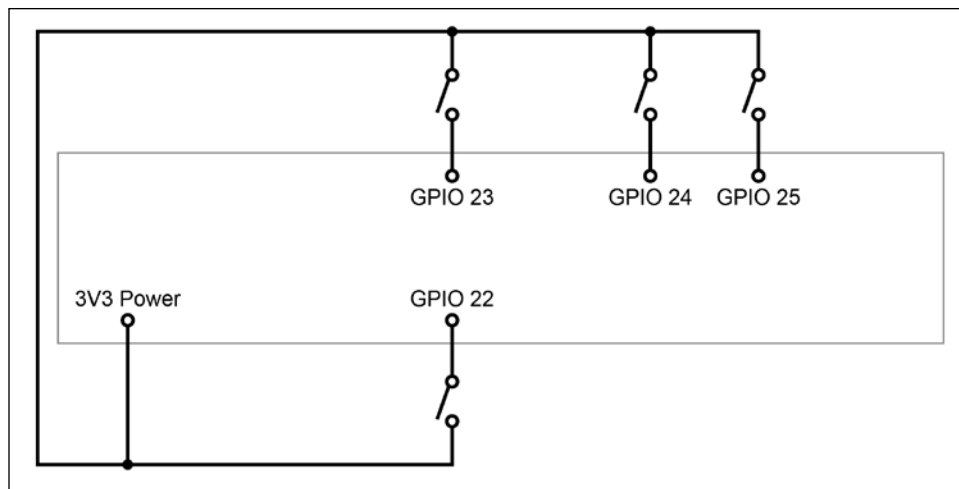
Now that the controller is ready, it's time to connect it to the Raspberry Pi. One of the things that makes the Raspberry Pi different from a normal computer is its set of **General-Purpose Input/Output (GPIO)** pins. These are the 40 pins (or 26 for older Raspberry Pi models) in the top-left corner of the Raspberry Pi, just above the logo. As the name suggests, they can be used for any purpose and are capable of both sending and receiving signals:

Raspberry Pi B+ and 2 only																			
5V Power	5V Power	Ground	GPIO 14	GPIO 15	GPIO 18	Ground	GPIO 23	GPIO 24	Ground	GPIO 25	GPIO 8	GPIO 7	ID_SC	Ground	GPIO 12	Ground	GPIO 16	GPIO 20	GPIO 21
3V3 Power	GPIO 0 (rev1) 2 (rev2)	GPIO 1 (rev1) 3 (rev2)	GPIO 4	Ground	GPIO 17	GPIO 21 (rev1) 27 (rev2)	GPIO 22	3V3 Power	GPIO 10	GPIO 9	GPIO 11	Ground	ID_SD	GPIO 5	GPIO 6	GPIO 13	GPIO 19	GPIO 26	Ground

The preceding figure shows what each of the pins does (if you have 26 pins, these are the ones present on the left-hand side of the preceding image). In order to create a useful circuit, we need to connect one of the power pins to one of the ground pins with some sort of electrical component in between. The GPIO pins are particularly useful because we can make them behave like either power or ground pins and they can also detect what they're connected to.

Note that there are two versions of the pin numbering system. You will almost certainly have a revision 2 Raspberry Pi. The revision 2 boards have mounting holes, while the revision 1 board has none (these holes are surrounded by metal and are large enough to put a screw through them. It's easy to spot them if they're there). It is safest to simply not use any of the pins that have different numbers in different revisions.

To connect your controller to the Raspberry Pi, connect one wire from each button to a 3.3V (3V3) power pin, and each of the remaining wires to a different GPIO pin (one with GPIO in its name, as shown in the previous figure). In my example, I will use pins 22, 23, 24, and 25. Everything is now connected, as shown here:



We're now ready to create our game!

Coding the game

Here's a quick recap of how this example game is going to work. The Raspberry Pi will choose a random button and ask the player to press it. Every time the player presses the correct button, they get a point, and every time they press a wrong button, they lose a point. Once the correct button has been pressed, the Raspberry Pi selects a new button as the target. The aim is to score as many points as possible in 30 seconds.

In Python 2, navigate to **File | New Window**. This will bring up a new empty **Code Editor** window, which is where all our code will go.

Random behavior

The first job, then, is to write some code that will choose a random button for the player to press. Take a look at the following code snippet:

```
import random
options = [22, 23, 24, 25]

def nexttarget():
    target = random.choice(options)
    print target
    return target
```

In the first line of the code, we import the `random` module, which we saw in the previous chapter, *Chapter 4, Creating Random Insults*.

In the second line of the code, we create a list of options. These are the GPIO pins that the buttons are connected to.

Next, we create a function called `nexttarget`. The empty brackets afterwards show that we do not need to pass any arguments to this function for it to work. The function chooses one of the options at random and stores it in a variable called `target`. We do this using the `random.choice` function, which we've seen before. We then print the target to display it to the player and return the target to whichever piece of code asked for it.

Type it all into the **Code Editor** and run it. Type `nexttarget()` next to the `>>>` marker in the Shell and press *Enter*, and you should see numbers being displayed. You can do this as many times as you like to make sure random pins are being displayed. The problem is that if the player is told to press pin 22, he/she might not know which button is being referred to. Let's change our code to improve that. Go back to the **Code Editor** and update your code, as follows:

```
import random
options = {22:"A", 23:"B", 24:"C", 25:"D"}

def nexttarget():
    target = random.choice(options.keys())
    print options[target]
    return target
```

The main difference is that we've changed `options` from a list to a dictionary (note the curly brackets being used instead of square ones). Using a dictionary allows us to give each pin a name, which will be more useful to the player. In this case, I have connected pin 22 to the A button and so on. Remember that in proper coding terms, the dictionary links each **key** (pin number) with a **value** (name). Our target pin must therefore be chosen from the dictionary's keys, so we add `.keys()` in the line where we choose a pin. Finally, when we display the target to the player, we get its name from the dictionary using the square brackets.

Using the controller

Next, we need to detect which button is currently being pressed. Look at the following code snippet (pay careful attention to the indentation):

```
import RPi.GPIO as GPIO

def buttonpressed():
    for pin in options.keys():
        if GPIO.input(pin) == GPIO.HIGH:
            return pin
    else:
        return None
```



Once again, we're importing a module that does some of the behind-the-scenes work for us. This time, we've used `as` to give it a slightly shorter name, which will hopefully make the rest of the code easier to read. We usually put all imports together at the very top of the code, as this makes it easier to see them all at once and makes them accessible for as much of the program as possible. All the code for this chapter is shown together toward the end, in case you are unsure where to put a particular piece of code.


Inside the `buttonpressed` function, we have a `for` loop. This is a lot like the `while` loop we've used before, except we tell it to stop after it has run a certain number of times. In this case, we tell it to run once for each of the pins in our dictionary of `options`.

We then check to see what signal that pin is receiving. If it is receiving `GPIO.LOW`, we know that the button is not being pressed, but if it is receiving `GPIO.HIGH`, we know that the button is being pressed and that there is a connection from this pin, through the button, to the power pin (in electronics, we say a signal is *low* if it is connected to the ground and *high* if it is connected to the voltage supply). If the button is being pressed, we return the pin number. Remember that a single equals sign is used to give a variable a new value, but a double equals sign checks to see if two values are the same. If none of the pins are being pressed, we return the special `None` value.

Before we can access the pins, we need to prepare them. Since they can be used for any purpose, we need to tell them what their job is for this particular piece of code. Add the following function to your code in the **Code Editor**:

```
def preparepins():
    GPIO.setmode(GPIO.BCM)
    for pin in options.keys():
        GPIO.setup(pin, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)
```

The `GPIO.setmode` line selects a particular numbering scheme for the Raspberry Pi's pins. Then, we have another `for` loop that looks at each of the pins in turn. For each pin, we choose `GPIO.IN` to say that it should be an input and receive signals and we use `GPIO.PUD_DOWN` to say that if nothing is connected to the pin, its signal should be pulled down to behave like `GPIO.LOW` (no button press).

 Pulling a pin in a particular direction prevents its value from floating and sometimes looking like `GPIO.HIGH` and at other times looking like `GPIO.LOW`.

This function will need to be run before we receive any signals from the pins in the `buttonpressed` function (if you do try to run this code now, you may get some strange error messages; we'll address these soon).

Adding a time limit

We can make sure that the `preparepins` function is always run before the `buttonpressed` function by writing it into our program. Let's now start building the function that brings everything together to create a game. For now, we want to set up the GPIO pins and make sure the game lasts the correct length of time, in this way:

```
import time

def play(duration):
```

```
preparepins()

start = time.time()
end = start + duration

while time.time() < end:
    # Do stuff
    time.sleep(0.1)
```

Once again, we are importing a module of existing code to do some of the hard work for us. This time, it's a module full of functions that deal with time and we are particularly interested in the one that tells us what the current time is.

Notice that we give `duration` as an argument to the `play` function. This lets us easily change the length of the game later, if we like. We then make absolutely sure that the `preparepins` function happens first by executing it straight away.

Next, we make a note of the current time using the `time.time()` function and store it in a variable called `start`. We calculate the time at which the game should end by adding the length of the game to the current time.

We then enter a `while` block (or `forever if`, if you prefer), which continues until the current time passes the time when the game should end. Inside the `while` block, we have a comment beginning with `#`. Comments are ignored by Python, but are useful for the programmer. You can leave notes for yourself to explain what a piece of code does. In this case, we've left a comment to say that there is more code to go inside, but we'll come back to it later. Finally, we put our program to sleep for 0.1 seconds. This has the following two purposes:

- It ensures that we don't waste time checking whether the buttons are pressed immediately after a previous check.
- It makes reading from the pins more reliable. In the instant after pressing a button, the paper clip may actually bounce up and down a tiny bit, making it seem like the button is being pressed multiple times. If this is the way this game works, the player could end up losing points, as the game may think the wrong button is being pressed.

Bringing it all together

Now, let's fill in the gaps and turn our program into a game. We want to use `nexttarget` and `buttonpressed` together to tell us whether the right or wrong button is being pressed and we want to keep track of the score. Update the `play` function so it looks like the following code snippet:

```
def play(duration):
    preparepins()

    start = time.time()
    end = start + duration
    score = 0

    target = nexttarget()
    while time.time() < end:
        button = buttonpressed()
        if button == target:
            score = score + 1
            print "Correct!"
            target = nexttarget()
        elif button != None:
            score = score - 1
            print "Wrong!"
        time.sleep(0.1)

    print "Your final score is", score
```

Here's a summary of what's changed:

- We've created a new variable called `score`, which starts at 0. Whenever the player presses the right button, the score goes up, and whenever they press the wrong button, it goes down. At the end of the game, we display the final score.
- We added another new variable called `target`. This is the pin connected to the button that we want the player to press. We set a target using `nexttarget` when the game first starts, and we update the target whenever the player presses the correct button.

- Inside the `while` block, we check which button is being pressed (if any). If the pressed button is the same as the target button, we give the player a point. Otherwise, if a different button is being pressed, we take a point away. `elif` is short for `else if`, and is used when we have multiple `if` blocks, but only want one of them to be executed. There's also a third possibility, that no button is pressed at all. In this case we want to do nothing at all, so we can leave the `else` block out entirely.

That's it! The game is ready to play. There's just one final small piece of code to add to the very end of the program, which could make things easier for us later:

```
if __name__ == "__main__":  
    play(30)
```

This is a special small trick that allows us to reuse our code later as its own module, or just play the game without having to load and run all of the code in Python 2 first.

Now, if you try to play the game, you will probably get an error message. This is because the Raspberry Pi's operating system wants to protect all of its hardware. You could end up doing dangerous things if you were allowed to change whatever you like! In this case, though, our actions are limited to the GPIO pins, so we can be fairly sure that we won't break anything as long as you have followed the instructions in this chapter carefully. Save your code and close down the Python Shell and Code Editor windows. Open **LXTerminal** and type in `sudo idle <name of your program>`. You might be asked to enter your password (the default is `raspberrypi`). You should see the windows open up again, and it should look exactly the same as before. This time, however, you should be able to navigate to **Run | Run Module** to play the game. The difference is the `sudo` command. This tells the Raspberry Pi that we know what we're doing and that we're sure we're not going to damage anything.



Be very careful when using the `sudo` command because the computer will always do exactly what you tell it to, even if this means causing permanent damage.

Our little extra piece of code at the end of the program gives us a different way of starting the game. Close down the Python Shell and Code Editor windows and type `sudo python <name of program>` into the terminal. The game should start much more quickly this time.

Complete code listing

The complete code listing section shows the complete program. This may be useful if you're not sure where the different code snippets should go, or if your program isn't working and you want to compare it to something that works:

```
import RPi.GPIO as GPIO
import random
import time

def preparepins():
    GPIO.setmode(GPIO.BCM)
    for pin in options.keys():
        GPIO.setup(pin, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)

def nexttarget():
    target = random.choice(options.keys())
    print options[target]
    return target

def buttonpressed():
    for pin in options.keys():
        if GPIO.input(pin) == GPIO.HIGH:
            return pin
    else:
        return None

def play(duration):
    preparepins()

    start = time.time()
    end = start + duration
    score = 0

    target = nexttarget()
    while time.time() < end:
        button = buttonpressed()
        if button == target:
            score = score + 1
            print "Correct!"
            target = nexttarget()
        elif button != None:
            score = score - 1
            print "Wrong!"
```

```
        time.sleep(0.1)

    print "Your final score is", score

options = {22:"A", 23:"B", 24:"C", 25:"D"}

if __name__ == "__main__":
    play(30)
```

The keyboard version

If you do not have access to the components necessary to create your own controller, here is a slightly modified program that uses the keyboard instead. You might notice that its structure is exactly the same as the previous program. Separating tasks into different functions allows us to make changes like these quickly and easily:

```
import pygame, pygame.event, pygame.key
from pygame.locals import *
import random
import time

def prepare():
    pygame.init()
    screen = pygame.display.set_mode((250, 1))
    pygame.display.set_caption("Test your speed!")

def nexttarget():
    target = random.choice(options.keys())
    print options[target]
    return target

def keypressed():
    pygame.event.pump()
    keypressed = pygame.key.get_pressed()
    for key in options.keys():
        if keypressed[key]:
            return key
    else:
        return None

def play(duration):
    prepare()

    start = time.time()
```

```

    end = start + duration
    score = 0

    target = nexttarget()
    while time.time() < end:
        key = keypressed()
        if key == target:
            score = score + 1
            print "Correct!"
            target = nexttarget()
        elif key != None:
            score = score - 1
            print "Wrong!"
        time.sleep(0.1)

    print "Your final score is", score

    pygame.quit()

options = {K_w:"w", K_a:"a", K_s:"s", K_d:"d"}

if __name__ == "__main__":
    play(30)

```

What's next?

Now that your game is working, you might like to try using most of the same code to create different games (I suggest that if you make changes, save it to a different file, so that you don't lose your current game). In particular, you could change `nexttarget()` so that it asks a question and gives some possible answers, and the player has to choose an answer as quickly as possible. Alternatively, you could create a *Simon Says* style game, where the game gives a sequence of buttons that must be pressed and the player tries to repeat it.

If you have an Internet connection and are feeling adventurous, you could try using your controller to play your Angry Birds game, as shown in *Chapter 3, Making Your Own Angry Birds Game*. Search the Internet for *ScratchGPIO* to download an enhanced version of Scratch, and try to explore how it can interact with the Raspberry Pi's GPIO pins.

If you're interested in learning more about electronics and what you can do with GPIO, take a look at Adafruit's online tutorials at <http://learn.adafruit.com/category/learn-raspberry-pi>.

Summary

In this chapter, we used the Python programming language to create a game. We created an electronic circuit to act as the game controller, and used code to detect when the buttons were being pressed. We revisited the basics of the Python language, and saw how separating the code into multiple functions makes it more flexible and easier to manage.

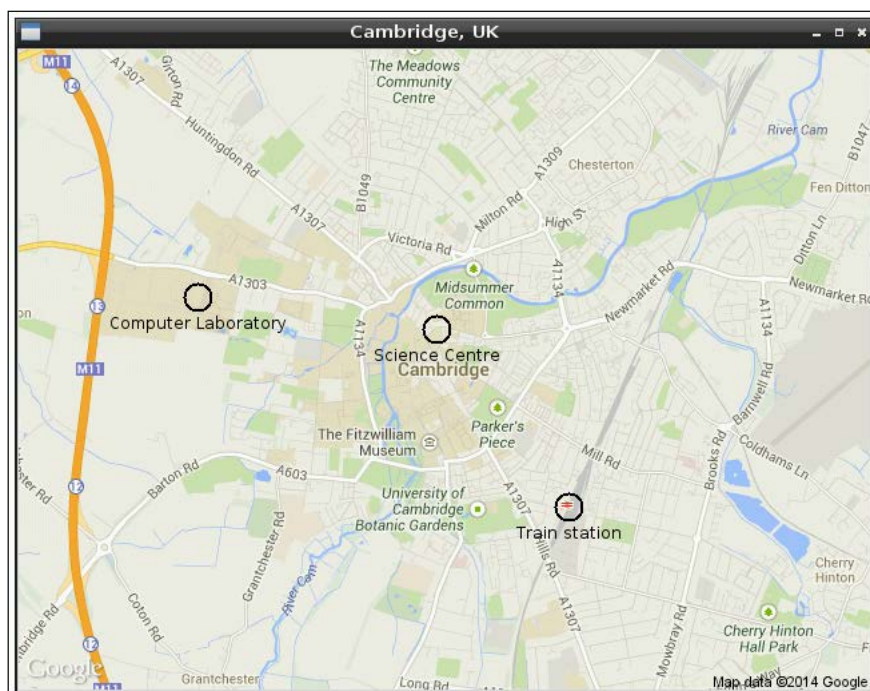
In the next chapter, we will build on this Python knowledge to create an interactive map.

6

Making an Interactive Map of your City

In this chapter, we're going to learn more about Python and its available modules by creating a program that will allow us to create notes on a map of our local area.

A program such as this needs a proper **Graphical User Interface (GUI)**, which is just a complicated way of saying that it is a visual program with things to see and buttons to click. Here's what the program looks like when it's finished:



As you can see, the program looks quite professional with its title bar and buttons. You will be able to click on locations on the map and give helpful labels. By the end of this chapter, you will know enough about building GUIs to be able to add all sorts of additional features.

Hello world!

As is traditional when we learn about a new technology, we're going to start with the simplest program possible, just to make sure that we understand the basics. In this case, we're going to create a basic window with a title and some text inside.

Tkinter

There are many different Python modules available that let us create graphical programs, but we're going to use one called **Tkinter**. This module is included with Python by default and works on almost all computers and operating systems. It allows Python to communicate with the **Tk** toolkit, and it is Tk which will generate our displays.

Tkinter has easy-to-use functions to create textboxes, buttons, scroll bars, menus, and more. Collectively, these components are called **widgets**. To create a graphical user interface, we combine a number of widgets with a **layout**, which tells Tkinter how the widgets should be arranged. For example, we could say that all the buttons should be placed in a row or that they should be arranged vertically.

A summary of how to use the available widgets in the *Extensions* section is provided at the end of this chapter.

Writing the program

Before we start, open a fresh Python 2 window and make sure you are in the **Code Editor** window (navigate to **File | New Window**).

1. The first thing we want to do is to import the Tkinter module so we can make use of all the functions it contains; therefore, add the following line of code at the top of the file:

```
import Tkinter
```

2. Now would also be a good time to save the program and give it a useful name. Navigate to **File | Save** and save the program as `hellogui.py`.

3. Creating a window is very simple. All we need is the following code:

```
window = Tkinter.Tk()
window.mainloop()
```

Leave a blank line below the import line (for neatness) and type in the preceding code. That's it!

4. We can now run the program (either by navigating to **Run | Run Module** or by pressing *F5*), and we will see an empty window appear.

The first line of code imports the Tkinter module, allowing us to access its functions; the second line creates a window; and the third line of code tells the window to enter its main loop. The main loop causes the window to be shown on the screen and lets it wait until any of its buttons are clicked (this is similar to how we had a loop in the previous chapter that waited for our controller buttons to be pressed.)

There are a few more very simple things we can do before we move on to creating the main program. This extra code must go between the previous two lines. The final `mainloop` line doesn't finish until the window is closed, so any code that comes afterwards will run too late to be shown on the screen. First, we can give the window a title, as follows:

```
window.title("Some text here")
```

We can give the window any title we like. Secondly, we're going to place a simple widget in the window that displays some text, as follows:

```
label = Tkinter.Label(window, text="Hello!")
label.pack()
```

This code is a little more complex. First, we create a **Label** widget—a widget to display text (or images). When we're creating it, we pass in two arguments: the window we've created and the text that needs to be shown. Note that the second argument has been given the name `text`, but we provide the existing window without giving it a new name. In Python, functions can have the option of receiving lots of different arguments. All of the compulsory ones come first and don't need to be given names; the function can tell them apart from the order they are in. After that come the optional arguments. We always need to give names to these so that the function can tell which arguments have been included and which have been left out. We need the first argument because the Label widget needs to know which window it will be in. In more advanced GUIs, we can even tell widgets which sections of the GUI they should go in. The second line of code `packs`, the Label widget to work out what size it is and start displaying it.

You might notice that if you run your program now that the window has shrunk to fit the Label widget we just added, so we can't see the title anymore! We can fix this by telling the window the minimum size it is allowed to be, as shown in the following code snippet:

```
width = 200
height = 50
window.minsize(width, height)
```

You might like to tweak the width if you have a long window title. You should now have a window that looks something similar to this:



Now that the window has a minimum size, you will notice that if you drag the edges of the window, you can make it larger; however, you can't make it any smaller.

We're now ready to move on. The following is the complete code for this simple example. I've grouped some of the lines together to keep things organized, but the main point is that anything involving the way the window looks is placed between the window being created and the window's main loop starting:

```
import Tkinter

width = 200
height = 50

window = Tkinter.Tk()
window.title("Some text here")
window.minsize(width, height)

label = Tkinter.Label(window, text="Hello!")
label.pack()

window.mainloop()
```

Getting a map

In this section, we're going to use **Google Maps** to get an image of our local area to display in our window.

No Internet? No problem!

Since Google Maps is an online service, an Internet connection is required to download a map. However, if your Raspberry Pi isn't connected to the Internet, there is still a way to proceed. Python is cross-platform. This means that it works on lots of different computers and operating systems. So long as you have access to another computer that does have an Internet connection, all the code in this chapter will work.

Python can be downloaded from <http://www.python.org/download/>, and the code in this book is based on Python 2.7 (Python is often preinstalled on Linux and Mac OS X operating systems, and it is best to keep it up to date with your built-in packaging system). Once installed on any computer, IDLE will be available and should behave exactly as Python 2 does on the Raspberry Pi.

Google Maps

Google has made it very easy to access its maps from the programs that we've written ourselves (up to 1000 times per day). All we need to do is create a web address with all the information we want about the map.

All addresses start with `https://maps.googleapis.com/maps/api/staticmap?` and contain all sorts of information, separated by `&` symbols after the question mark:

- `center=location`: This is some text describing the location that the map should show. It could be a town name, postal code, or the name of a road or building. Web addresses should not contain any spaces, so if your chosen location does have spaces, they should be replaced by the `+` symbol, or the special `%20` sequence.
- `zoom=value`: This is a number that increases as we zoom in to the map. A value of around 13 to 14 seems to give good results for this project, but you might like to try other values.
- `size=widthxheight`: These are values in the form of pixels. In this chapter, I'm going to use a width of 640 pixels and a height of 480 pixels.
- `format=type` (optional): This denotes the format of the image to be downloaded, such as JPEG, GIF and PNG (default). In this chapter, we're going to use GIF as it works best with Tkinter.

- `maptype=type` (optional): This tells us what view of the map we should get. Do we want a `satellite` image or a `roadmap`, or do we want to see the `terrain`? If we don't choose a map type, we will get a road map.
- `sensor=true/false`: This tells us if we are using GPS (or something similar) to choose the location. For this project, it will always be set to `false`.

A full list of available options and their explanations can be found online at https://developers.google.com/maps/documentation/staticmaps/#URL_Parameters.

So, an example web address might be `https://maps.googleapis.com/maps/api/staticmap?center=Cambridge,%20UK&zoom=13&size=640x480&format=gif&sensor=false`. Here, I have chosen the map of Cambridge, UK, with a zoom level of 13, and an image that is 640 x 480 pixels and in the GIF format. You may want to type this address into a web browser and play with the various options to see what's possible.

Generating the address

So, how do we create these long web addresses automatically in our program? It turns out that Python makes this very easy for us with its `format` function. The `format` function takes some text and looks through it for markers that look like `{0}`, where 0 can be any number. Whenever the function sees one of these markers, it replaces it with the argument at that position, as shown in the following examples:

```
>>> "{0}".format(14)
'14'

>>> "Second = {1}, first = {0}".format(1,2)
'Second = 2, first = 1'
```

The main thing to look out for is that programming languages like to start counting from zero, so if you want to access the first argument of `format`, you use `{0}`, and if you want the seventh argument, you use `{6}`.

To generate our address, we can use the following code snippet:

```
address = "http://maps.googleapis.com/maps/api/staticmap?\ncenter={0}&zoom={1}&size={2}x{3}&format=gif&sensor=false"\n.format(location, zoom, width, height)
```

This is just a slightly longer and more complex version of what we've seen already. The `\` symbols allow us to break the line into multiple parts so it doesn't go off the edge of the screen (or page), and they do not show up in the final address if we start a new line immediately after the `\` symbol.

To make our code more readable and useful, it is best if we put this address creation code in a separate function. That way, we can generate addresses any time we like when the program is running, without having to copy the code.

Place the following code immediately after the `import` statement in `hellogui.py`, and then save it to a new file called `mapping.py`:

```
import urllib
def getaddress(location, width, height, zoom):
    locationnospaces = urllib.quote_plus(location)
    address = "http://maps.googleapis.com/maps/api/staticmap?\
center={0}&zoom={1}&size={2}x{3}&format=gif&sensor=false"\
.format(locationnospaces, zoom, width, height)
    return address
```

You'll notice that there's an extra line at the start of the function that uses `urllib.quote_plus` to make sure that there are no spaces in the name of the location by replacing them with `+` symbols. It can also handle any other characters that aren't allowed in web addresses. We had to import the `urllib` module first to get access to this function. The `urllib` module is short for URL library and allows us to access information over the Internet. **Uniform Resource Locator (URL)** is just another name for a web address. You may want to provide extra options to add extra arguments to the function later.

We can now see if our code works. Run the program and close the window that pops up – we're not interested in it for the moment. In the Shell (next to the `>>>` marker), type in `getaddress("Cambridge, UK", 640, 480, 13)`, press *Enter*, and check that the link is the same as shown in the earlier example. You can even paste it (without the surrounding quotation marks) into a web browser to check that it works.

If you're really keen on controlling things with code, try out the following code snippet to see what it does. You can type this in as part of your program, or type it into the Shell after you have run the program in the Code Editor:

```
import webbrowser
webbrowser.open(getaddress("Cambridge, UK", 640, 480, 13))
```

Downloading an image

Now that we can create a web address for our map, we want to download the image to use in our program. To do this, we're going to create another function called `getmap` that uses `getaddress`. Here's the code snippet:

```
import base64
def getmap(location, width, height, zoom):
```

```
address = getaddress(location, width, height, zoom)
urlreader = urllib.urlopen(address)
data = urlreader.read()
urlreader.close()
base64data = base64.encodestring(data)
image = Tkinter.PhotoImage(data=base64data)
return image
```

We first need to import another module. The `base64` module allows us to convert the downloaded image data into something that Tkinter can use.

The first thing we do in our new function is create an address using the previous function. We can then connect to this address using `urllib.urlopen` and download the data using `read`. We make sure to tidy up afterwards by using `close`. (The `urlreader` object might have used some temporary storage that is no longer needed now that we have the data).

Unfortunately, the data we downloaded isn't in a form that Tkinter can use, so we need to convert it using `base64.encodestring`. You don't need to understand how this works; just be aware that it's there. (If you're interested in what's going on inside the module, take a look at <http://docs.python.org/2/library/base64.html>.) Finally, we convert the data into an image using `Tkinter.PhotoImage` and return it.

Using an image

We now have an image ready to use, so it's time to display it in our program. It is possible to put the image inside the Label widget that we already have, but we will want to draw on top of it later, so we will use a **Canvas** widget instead. You can think of a Canvas widget as a bit like the canvas an artist would use. It allows us to draw all sorts of shapes and text in any color we like. For now, we're just going to draw our map.

Replace the two lines of code that mention the Label widget with the following code:

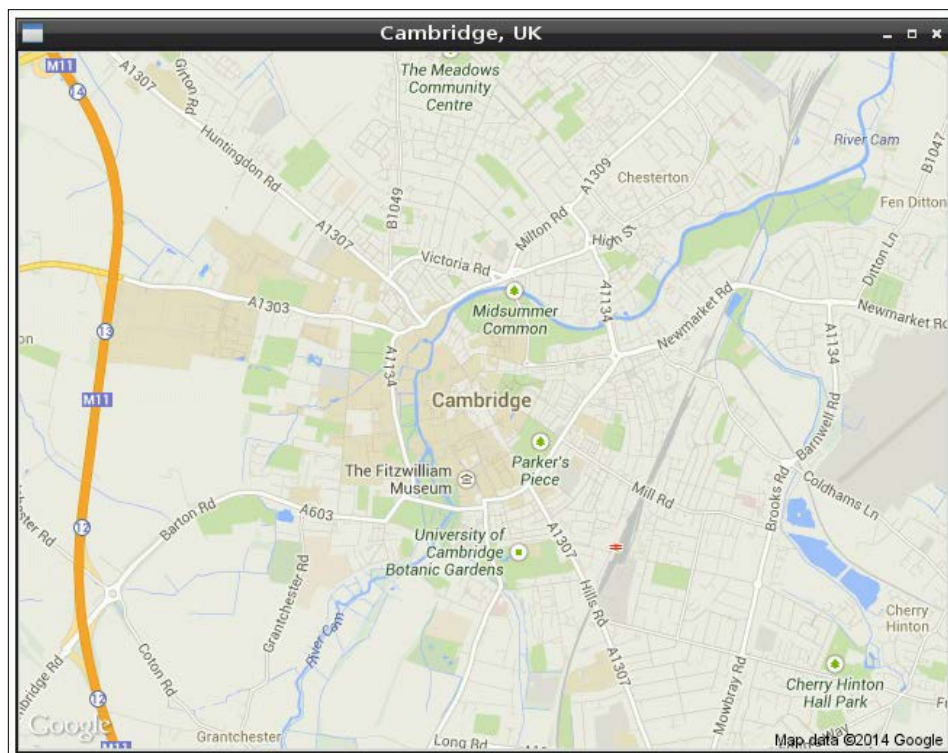
```
mapimage = getmap(location, width, height, zoom)
canvas = Tkinter.Canvas(window, width=width, height=height)
canvas.create_image(0,0,image=mapimage,anchor=Tkinter.NW)
canvas.pack()
```

First, we get the image using our `getmap` function. We then create a Canvas inside our window with a particular width and height. Then, we draw our image on the Canvas. We say that we want the northwest (NW) corner of the image to be placed at $(0, 0)$ coordinates within the Canvas. Since the northwest and $(0, 0)$ coordinates both mean the top-left corner, and the Canvas is the same size as the image, the image will fill the Canvas exactly. Finally, we pack the Canvas widget as we did with the Label widget.

One last thing to do is giving some sensible values for `location`, `width`, `height`, and `zoom`. We already have values for the width and height, but we'd like our map to be a little larger than our previous window. Replace the old `width =` and `height =` lines with the following code snippet:

```
location = "Cambridge, UK"
width = 640
height = 480
zoom = 13
```

Feel free to experiment with different values. When you run your program, you should now see something similar to this:



I have also updated the title of the window to use `location` rather than "Some text here" from before. If your code isn't working, it's likely that there's either a spelling mistake somewhere, some code needs to be moved up or down within the program, or Python isn't sure which blocks of code are meant to be inside other blocks. Remember that the number of spaces at the beginning of each line is very important. The recommendation is to use four spaces for each level of indentation. For example, the very first line of a function (in this project) should have no spaces in front of it, code inside this function should have four spaces at the start of each line, and code inside an `if` or `while` block inside that function should be indented with another four spaces. If you're stuck, take a look at the code listing near the end of the chapter.

Adding markers

The next thing we want to do is add a marker to the map whenever we click on it with the mouse. This can be done in two parts: by detecting the click and reacting to the click.

Detecting mouse clicks

Detecting mouse clicks is very simple. Tkinter does most of the work for us. All we have to do is **bind** a function to the mouse button. Once the program has entered its main loop, whenever the mouse button creates an event (by being clicked on), the function will be executed. Reacting to an event in this way is similar to using a `when key pressed` code block in Scratch. Place the following line of code with the rest of the Canvas code before the main loop:

```
canvas.bind("<Button-1>", canvasclick)
```

This code says that whenever `Button-1` (the left mouse button) is clicked on, run the `canvasclick` function. We'll write this function next.

We can create these bindings for as many buttons and keys as we like and for any widget that we like. The `"<Button-3>"` button is the right mouse button, `"<space>"` is the Space bar, `"<Return>"` is the *Enter* key, and `"a"`, `"b"`, `"c"`, and so on correspond to the letters. There are even events called `"<Enter>"` and `"<Leave>"` that can tell when the mouse moves over the widget.

Reacting to mouse clicks

When the mouse button is clicked, an event is given to our `canvasclick` function. The event contains lots of information, including the position of the click, the widget that was clicked, and the key that was pressed (if any).

Here's a quick version of `canvasclick` that should let you make sure that mouse clicks are being detected properly. Place it beneath the `getmap` function as follows:

```
def canvasclick(event):  
    print "Mouse click at position", event.x, event.y
```

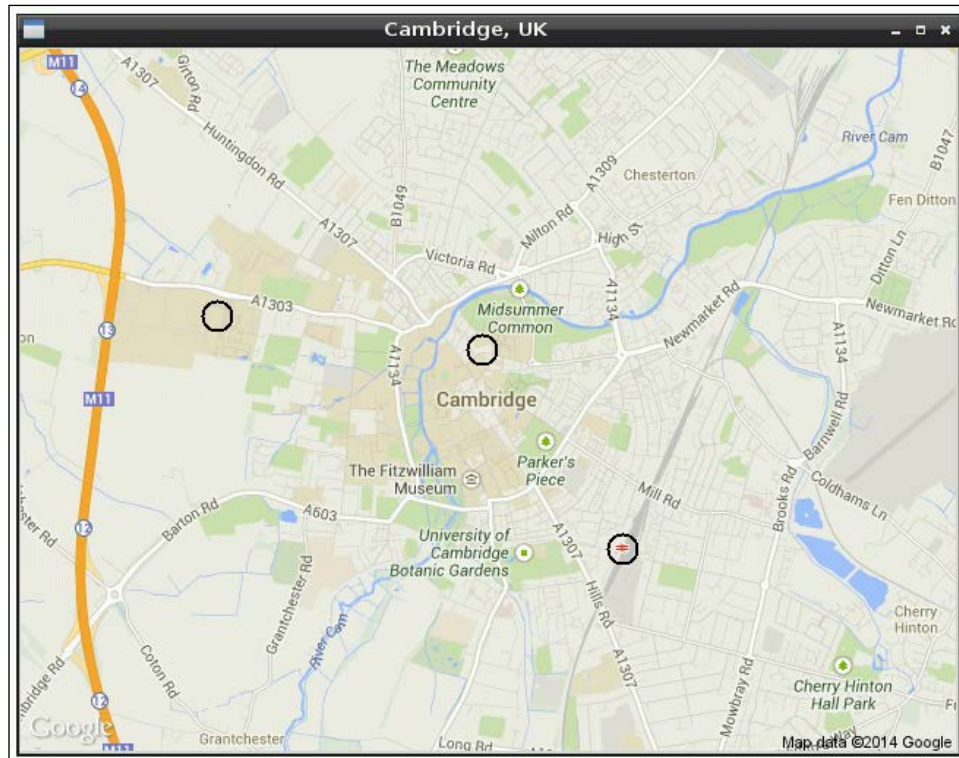
When we run the program, we should now see pairs of numbers being displayed in the Shell whenever we click on our map. These numbers should change depending on where we click on the map. What we really want, though, is to draw a marker on the map so we can highlight interesting points. We'll replace the `print` line with the following code, and a circle will be drawn on the map at each position clicked:

```
x,y = event.x, event.y  
widget = event.widget  
size = 10  
widget.create_oval(x-size, y-size, x+size, y+size, width=2)
```

We're going to use `event.x` and `event.y` a few times, so here we've given them the more convenient names of `x` and `y`. We've done the same thing for `event.widget` (the widget that received this mouse click event), giving it the more convenient name of `widget`. The `size` variable stores the distance in pixels, from the click position to the edge of the circle, that is, the radius of the circle. You can change it if you like.

Finally, we draw the circle using `widget.create_oval`. The first four arguments are the coordinates of the left, top, right, and bottom edges of the circle, and `width` is the width of the line used to draw the circle. You can add extra arguments, such as `outline="red"` to change the color of the line and `fill="blue"` to change the internal color. I particularly like the `activeoutline` and `activefill` arguments, which work in the same way but only show their colors if the mouse is over the marker. Experiment until you have a marker design you like.

You should now have a program that looks like something similar to the following screenshot:



Adding labels

It would be useful if whenever we clicked on the map, along with adding a circular marker, we could also add a few words to describe what we're marking.

Basic labels

Getting some text from the program's user is going to be slightly complex, so let's create a simple version first to make sure we have the right code structure. Add the following two lines of code right at the end inside the `canvasclick` function:

```
label = getlabelname()
widget.create_text(x, y+2*size, text=label)
```

The first line of code gets some text from a function that we haven't written yet called `getlabelname`. This function will eventually ask the user to type some text into a small pop-up window, but for now, it will just give us a default message. The second line of code draws our text at a particular position just underneath the circle. As with `widget.create_oval` earlier, `widget.create_text` allows the text color to be set using the extra arguments of `fill="colour"` and `activefill="colour"`.

Here is our most basic version of the `getlabelname` function. We will flesh it out in the next section. Since it is used in `canvasclick`, `getlabelname` needs to be placed somewhere before it in the program. Putting `getlabelname` immediately above `canvasclick` is a good idea because the two functions are used together, and this way, we can see both of them in the **Code Editor** at the same time:

```
def getlabelname():
    text = "This is a label"
    return text
```

When you run your program, you should now see small text labels appear below the markers whenever you click on the map.

Pop-up windows

Let's now make `getlabelname` a little more interesting. We're going to open a new window that asks users to give a name for their marker. This window should have an instruction for the users telling them what to do, a place for the users to type their marker's name, and a button to click when they're finished.

First, we'll create a new window in a similar way to how we made our main window. Add the following code at the beginning of `getlabelname`, somewhere before the `return` line (there's a complete copy of this function at the end of this section if you're not sure where a particular piece of code should go):

```
popup = Tkinter.Toplevel()
popup.title("New marker")
popup.wait_window()
```

This time, we're using `Tkinter.Toplevel` instead of `Tkinter.Tk`. We only use `Tk` for the main window, and use `Toplevel` for all the others. The `wait_window()` method then behaves like `mainloop()`, and waits until the window is closed.

Next, we'll add a label with the instruction for the user. Remember that all the contents of a window must be created after the window is created but before we start its main loop. Type the following lines of code immediately above `popup.wait_window()`:

```
label = Tkinter.Label(popup, text="Please enter a label for  
your marker")  
label.pack()
```

The code so far is very similar to the very first window we created at the beginning of the chapter. You might want to try running the program and make sure that the new window does appear whenever you click on the map, and that the default label appears.

Next, we're going to add a textbox for the user to type into, as follows:

```
labelname = Tkinter.StringVar()  
textbox = Tkinter.Entry(popup, textvariable=labelname)  
textbox.pack()  
textbox.focus_force()
```

There are a couple of new things here. First, we create a **StringVar** called `labelname`. **StringVar** is short for *String Variable*, and *string* is another word that programmers use for *text*. So, `labelname` is going to hold a text variable for us. Second, Tkinter's name for a textbox is **Entry**. This reflects the fact that we can enter text into the box, rather than simply view the text that is already there. We pass our variable to the **Entry** when it is created. Now, we can access the text in the Entry through our variable—we'll get to this soon. As usual, we pack the **Entry** to prepare it to be displayed. Finally, we use `focus_force` to make sure that the textbox is the thing that has the user's attention. The pop-up window will now be the active window, and the textbox will be ready to type into. Without this line of code, the user would have to click on the textbox themselves before they could type anything in.

Next, we're going to add a button. When the button is clicked, the pop-up window should close, and we'll be ready to get the message out of the textbox. Here's the code we need:

```
button = Tkinter.Button(popup, text="Done")  
button.pack()
```

This simply creates a new button that says **Done** in our new window. With this in place, the pop-up window should look finished. If you test your code now, you should see this:



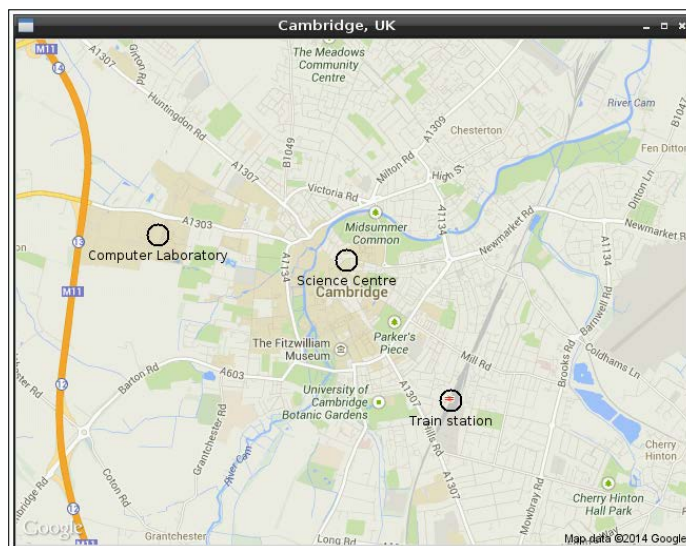
However, you'll notice that the button doesn't actually do anything yet. We need to give it a **command**. Update the button creation line as follows:

```
button = Tkinter.Button(popup, text="Done", command=popup.destroy)
```

Our window called `popup` has a function called `destroy` which closes the window. When the button is clicked, we want this function to be executed, so the window closes and we can retrieve the label name that the user typed in. To do this, we pass in the function as an extra argument when we create the button. Finally, to get the label name, replace the existing `text =` line with the following line of code just before the `return` line:

```
text = labelname.get()
```

That's it! You should now be able to click on the map, type in a label name, and see it appear when you click on **Done**. Your running program should now look like something similar to the following screenshot:



This is what your completed `getlabelname` function should look like:

```
def getlabelname():
    popup = Tkinter.Toplevel()
    popup.title("New marker")
    label = Tkinter.Label(popup, text="Please enter a label for your
marker")
    label.pack()

    labelname = Tkinter.StringVar()
    textbox = Tkinter.Entry(popup, textvariable=labelname)
    textbox.pack()
    textbox.focus_force()

    button = Tkinter.Button(popup, text="Done", command=popup.destroy)
    button.pack()

    popup.wait_window()

    text = labelname.get()
    return text
```

Code listing

Here is the complete code for the project used in this chapter. It can be used if you're getting strange error messages and want to compare your code with something that is known to work. It can also help you see what order the various snippets of code should be in.

The very first thing in the file should be the `import` statements. It's a good idea to put these in alphabetical order so that we can search through them more quickly when we import a lot of modules; this is shown in the following code snippet:

```
import base64
import Tkinter

import urllib
```

Next, we have two functions that work together. The first one creates a web address and the second downloads the map image from this address, as shown in the following code snippet:

```
def getaddress(location, width, height, zoom):
    locationnospaces = urllib.quote_plus(location)
    address = "http://maps.googleapis.com/maps/api/staticmap?"
```

```

center={0}&zoom={1}&size={2}x{3}&format=gif&sensor=false"\
.format(locationnospaces, zoom, width, height)
    return address

def getmap(location, width, height, zoom):
    address = getaddress(location, width, height, zoom)
    urlreader = urllib.urlopen(address)
    data = urlreader.read()
    urlreader.close()
    base64data = base64.encodestring(data)
    image = Tkinter.PhotoImage(data=base64data)
    return image

```

Then, we have the functions to deal with the pop-up window, which collects the label to give to a marker on the map. The first function tells the window what to do when **Done** is clicked on and the second then uses this function when it builds the window, as shown in the following code snippet:

```

def getlabelname():
    popup = Tkinter.Toplevel()
    popup.title("New marker")
    label = Tkinter.Label(popup, text="Please enter a label for your
marker")
    label.pack()

    labelname = Tkinter.StringVar()
    textbox = Tkinter.Entry(popup, textvariable=labelname)
    textbox.pack()
    textbox.focus_force()

    button = Tkinter.Button(popup, text="Done", command=popup.destroy)
    button.pack()

    popup.wait_window()

    text = labelname.get()
    return text

```

We then have the function that is executed whenever the map is clicked. This makes use of the preceding functions, as follows.

```

def canvasclick(event):
    x,y = event.x, event.y
    widget = event.widget
    size = 10

```

```
widget.create_oval(x-size, y-size, x+size, y+size, width=2)
label = getlabelname()
widget.create_text(x, y+2*size, text=label)
```

Finally, we have the following code that has to be executed when we first run the program (this function is traditionally called `main`):

```
def main():
    location = "Cambridge, UK"
    width = 640
    height = 480
    zoom = 13

    window = Tkinter.Tk()
    window.title(location)
    window.minsize(width, height)

    mapimage = getmap(location, width, height, zoom)
    canvas = Tkinter.Canvas(window, width=width, height=height)
    canvas.create_image(0,0,image=mapimage,anchor=Tkinter.NW)
    canvas.bind("<Button-1>", canvasclick)
    canvas.pack()

    window.mainloop()

if __name__ == "__main__":
    main()
```

Extensions

There are lots of things we could do now that we have a basic working GUI. Here are a few possible ideas:

- Adding buttons to zoom in or out
- Adding a textbox and button to update the location
- Adding a way to select different styles of map marker
- Selecting whether the map is a satellite image or a road map
- Saving and loading the map settings (the location, position of markers, labels, and so on)
- Allowing markers and their labels to be changed after they have been created

Complete details on how to use Tkinter can be found online at <https://wiki.python.org/moin/TkInter>.

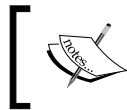
Layout

In this chapter, we have used only the `pack` layout, but there are also other ways to tell Python where you want your widgets to be displayed.

The `pack` layout is useful to fill the screen with a single widget (similar to our map) or to place widgets in a line (similar to our window when we type in label names).

The `grid` layout allows us to line up widgets both vertically and horizontally. All the widgets that we put in the same column form a vertical line, and all the widgets in the same row form a horizontal line. If no row or column is given, Python will put the widget in the first available place it finds. We can also have a widget reach across (or span) multiple rows or columns. Try replacing the three `.pack()` lines in `getlabelname` with the following lines of code:

```
label.grid(columnspan=2)
textbox.grid(column=0, row=1)
button.grid(column=1, row=1)
```



The `pack` and `grid` layouts do not work together. If you would like to use one of these layouts, you will need to make sure that the same layout is used for every widget.

There is also a third option, `place`, which allows us to set the exact position of the widget. This isn't used often because `pack` and `grid` do such a good job, and it has too many necessary arguments to summarize here.

Additional widgets

The next few sections give some very short code snippets, showing how widgets that we haven't covered in this chapter can be created. If you want to test them out, put the code just before the `window.mainloop()` line in your program. The new widget will usually appear just below the map when you run the program. If you run out of space on your screen, try reducing the height of the map to make more space.

Checkbutton

`Checkbutton` can either be empty or contain a check (tick). The following shows you the code snippet for the `checkbutton`:

```
state = Tkinter.StringVar()
checkbutton = Tkinter.Checkbutton(window, text="Button",
    variable=state, onvalue="checked", offvalue="unchecked")
checkbutton.pack()
```

Along with the button, we also need a `StringVar` variable (which is a text variable). The button has a particular value when it is on (`onvalue`) and a particular value when it is off (`offvalue`). These values are stored in the `StringVar` variable. To access the current state of the button, use `state.get()`.

Frame and LabelFrame

Frames and `LabelFrame`s simply contain other widgets. They allow us to structure lots of widgets better. A `Frame` is a plain container and a `LabelFrame` adds an outline and a label, as shown in the following code snippet:

```
labelframe = Tkinter.LabelFrame(window, text="LabelFrame")
button = Tkinter.Button(labelframe, text="Button")
button.pack()
labelframe.pack()
```

As you can see, we add `Button` to `LabelFrame` in the same way we would add it to a window, by passing `LabelFrame` as the first argument when we create the button.

Listbox

`Listbox` has a different option on each line. Options can be selected and deselected by being clicked on. Let's have a look at the following code snippet:

```
options = Tkinter.StringVar()
options.set("Option1 Option2 Option3")
listbox = Tkinter.Listbox(window, listvariable=options)
listbox.pack()
```

Along with `Listbox`, we also need a `StringVar` variable to hold the available options. Each option is separated by a space. We can access the number of the current selection using `listbox.curselection()` (remember that programmers like to count from zero, so the first option is at position 0.)

Menu

`Menu` contains several different options, and some kind of action is taken when an option is clicked:

```
topmenu = Tkinter.Menu(window)
dropmenu = Tkinter.Menu(topmenu)
window["menu"] = topmenu
topmenu.add_cascade(label="Menu", menu=dropmenu)
dropmenu.add_command(label="Option1", command=function1)
dropmenu.add_command(label="Option2", command=function2)
```

Here, we are creating two menus. The first (`topmenu`) goes across the top of the screen. The second (`dropmenu`) drops down from the top menu when it is clicked. The `topmenu` can contain any number of drop-down menus, these are added using `topmenu.add_cascade`. The `dropmenu` can contain any number of options, these are added using `dropmenu.add_command`. A different function is executed when each of the options is clicked (I've just used the names `function1` and `function2` as examples. You will need to actually name the functions in your program).

Menubutton

The `Menubutton` option is very similar to the `dropdownmenu` option that was used in the previous section, except that it is positioned as a button instead of being positioned within another menu at the top of the window. Take a look at the following code snippet:

```
menubutton = Tkinter.Menubutton(text="MenuButton")
menu = Tkinter.Menu(menubutton)
menubutton["menu"] = menu
menu.add_command(label="Option1", command=function1)
menu.add_command(label="Option2", command=function2)
menubutton.pack()
```

Message

`Message` is a lot like `Label`, which we have already seen, except that it is designed for longer pieces of text and can spread across multiple lines, as shown in the following code snippet:

```
message = Tkinter.Message(window, text="This is a message")
message.pack()
```

OptionMenu

`OptionMenu` gives a drop-down list, allowing the user to select one of the fixed number of options, as shown in the following code snippet:

```
state = Tkinter.StringVar()
optionmenu = Tkinter.OptionMenu(window, state, "Option1", \
                                "Option2")

optionmenu.pack()
```

We need a `StringVar` variable to hold the current selection, and this selection can be accessed using `state.get()`.

Radiobutton

Radiobuttons are usually used in groups, and only one can be selected at a time, as shown here:

```
state = Tkinter.IntVar()
radiobutton1 = Tkinter.Radiobutton(window, text="Option1", \
                                   value=1, variable=state)
radiobutton2 = Tkinter.Radiobutton(window, text="Option2", \
                                   value=2, variable=state)

radiobutton1.pack()
radiobutton2.pack()
```

We need a variable to hold the current selection. This time we're using an `IntVar` variable (an integer, which is a whole number variable), and each button has a value that will be stored in the variable when that button is selected. The key to only having one radio button selected at a time is to give the whole group the same variable argument. The current selection can be accessed using `state.get()`.

Scale

Scale gives a slider that can be used to choose a value between two limits, as shown here:

```
state = Tkinter.IntVar()
scale = Tkinter.Scale(window, label="Scale", from_=0, to=10, \
                      variable=state)

scale.pack()
```

We need `IntVar` (a whole number variable) to hold the current value, and we can choose the smallest and largest possible values using the `from_` and `to` arguments. We can get the current value of `Scale` using `state.get()`.

Spinbox

Spinbox is a box containing a number. Next to the box are two small arrow buttons that make the number larger or smaller, as shown here:

```
spinbox = Tkinter.Spinbox(window, from_=0, to=100, increment=10)
spinbox.pack()
```

We choose the smallest and largest possible values for `Spinbox` using the `from_` and `to` arguments, and we choose how much the value should change by when a button is pressed using `increment`. We can get the current value using `spinbox.get()`.

Summary

In this chapter, we learned how to make a GUI in Python. We learned how to create all sorts of different widgets that let the GUI do interesting things, and we also learned how to react to events, such as mouse buttons being clicked.

In particular, we created a mapping program that lets us click on the map to mark points of interest and even add useful descriptions for the markers. We have the knowledge and skills to add many extra features to our program by continuing to add buttons and other widgets.

In the next chapter, we'll explore how we can create music with code.

7

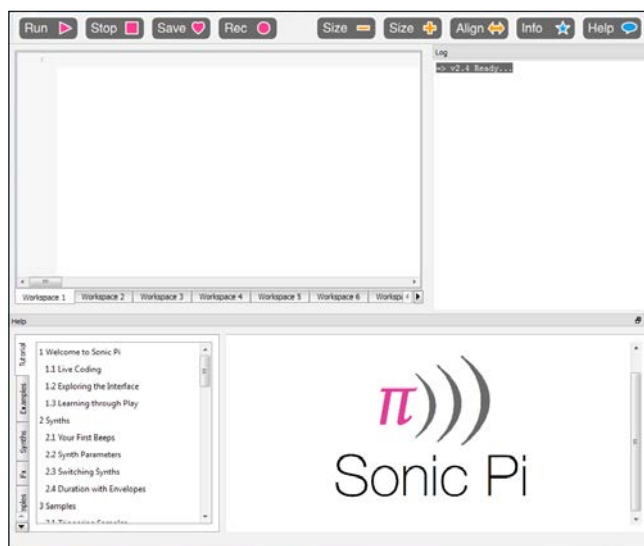
Building Beats with Sonic Pi

In this chapter, we're going to use a new application called **Sonic Pi** to create music using code. We will see how it's possible to create tunes and how the power of programming allows us to experiment and change things much more easily than we can with physical instruments.

This chapter is centered on sound, so you will need some speakers or headphones that can plug into the Raspberry Pi's audio port, or an HDMI monitor with built-in speakers.

Sonic Pi

Sonic Pi is a program designed specially for the Raspberry Pi. It allows us to create music by writing code. You can run Sonic Pi by choosing **Sonic Pi** in the **Programming** menu on the Raspberry Pi's desktop. Here's what you'll see:



Right at the top, there are all the buttons for controlling Sonic Pi's behavior. We can start and stop a tune playing, save our code, and record our own sounds. We can also decrease or increase the size of the text that our code is written in and neaten it up.

Below those buttons, on the left is the Code Editor, where we will type all of our code. Note that there are multiple **Workspace** tabs. These allow us to have multiple programs open at a time and switch between them easily. On the right is the **Log**. This will display information about every note that is played so that you can see what Sonic Pi is doing.

At the bottom is the **Help** system. This has example code and descriptions of all the sounds and programming features available.

Sonic Pi uses its own text-based programming language. It is similar to Python in some ways and different in others, but all that is important is that we will be doing the same basic operations. We will see throughout the rest of the chapter that all the languages we have seen so far share a core set of features and these features are common to almost every programming language that exists. If you can understand how these features work, all you need to do to learn a new language is learn how that language presents those features.



Sonic Pi is installed as part of the default Raspbian operating system on the Raspberry Pi and is also available on Windows and Mac OS X. It can be downloaded from <http://sonic-pi.net/>.

Getting started with Sonic Pi

Getting Sonic Pi to create a sound is very simple. Type the following code in the Code Editor and click on **Run**:

```
play 60
```

You should hear a tone.



If you do not hear anything, verify that your speakers are switched on and that the sound is not muted.

Here, we represent different notes as numbers; this is a convenient representation that the Raspberry Pi understands. A higher number represents a higher note. Try it for yourself; change the number and click on **Run**.

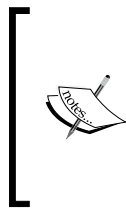
If you know a little about music, you might be familiar with the letter names of notes. Sonic Pi knows these too! Try this code:

```
play :C
```

This is the same note as before, but now it has a name. Any note from `A` to `G` will work, and you can also place a `b` or `s` after the name to make the note *flat* (lower) or *sharp* (higher).

Finally, we can put a number at the end of the note name to say which *octave* the note is in. Again, a higher number will give a higher pitched sound. Here's an example note that combines all the previously shown features:

```
play :Fs4
```



An octave is a range of sounds where the lowest pitch is exactly half that of the highest pitch. The repeating sequence of keys on a piano represents different octaves. An octave covers the notes A, B, C, D, E, F, and G; as well as A# (or Bb), C# (or Db), D# (or Eb), F# (or Gb), and G# (or Ab). Two notes that are an octave apart tend to sound good when played together.

Part of the beauty of representing notes as numbers in Sonic Pi is that we do not have to stick to whole numbers, as we do with many physical instruments. We can play note 60.5 or 71.419, or any other number we like.

Creating a tune

Any real piece of music is going to contain more than one note. Type the following code in the Code Editor and click on **Run**. What do you hear?

```
play 60
play 65
play 72
```

That might not have sounded the way you expected! All the notes played at the same time. Sonic Pi will play every note it sees until it reaches a `sleep` command, upon which it waits for a certain amount of time and then continues. This is useful when we want to play multiple notes at once, but to create a tune, we need them all to be separate. Put a `sleep` command after each `play` command, like this:

```
play 60
sleep 1
play 65
```

```
sleep 1
play 72
sleep 1
```

That's better! All the notes now play one after another. The number after the `sleep` command is the number of seconds to wait before playing the next note (or notes). Again, we can use any number we like, and the number can be different for each `sleep` command.

Now, music is largely based on repetition, so let's learn how to create loops in Sonic Pi. It's very simple—add a `loop do` and an `end` around the code that we want to repeat, like this:

```
loop do
  play 60
  sleep 1
  play 65
  sleep 1
  play 72
  sleep 1
end
```

The `do` and `end` tell Sonic Pi which lines of code are in the code block and the `loop` tells it what should be done with the block. In this case, we want the code to repeat forever. We don't need to indent the code within the block (as we do in Python), but it helps make the code more readable, especially when we have complex structures such as loops within loops. At any time, you can click on the **Align** button at the top of the window and it will choose what it thinks is the best indentation for your code.

If you like to repeat the code a fixed number of times, replace `loop` with any number followed by `.times`. For example, the first line can be `10.times do` if you want to repeat the code ten times.

Finally, let's tweak our code to create a random tune. The first thing we're going to do is put all the notes we want to play into a list. We can do this in exactly the same way as we did in Python. Here's a list that contains the three notes from the previous example:

```
[60, 65, 72]
```

See how we separate each number in the list with commas, and we wrap the whole thing with square brackets. You can change any of these numbers or add new ones if you like. To get a random number from our list, we will use the `choose` function that Sonic Pi provides. Here is a new code block to put inside our loop:

```
play choose([60, 65, 72])
sleep 1
```

If you run this code, you should hear a sequence of random notes taken from the list.

New sounds

Up until now, we've been using the default sound in Sonic Pi, called `beep`, but there are many other sounds available.

Open up a couple of empty lines before the loop, type in `use_synth`, and then press the spacebar. Synth is short for **synthesizer**, which is a program or machine that generates sound. You will see a long list of sound names appear, with `beep` at the top. Choose one of the sounds by double-clicking on it or typing enough of its name so that it is the only option. Then press *Enter*. If you run your code now, you will hear the same notes, but the sounds will be different. Try a few different sounds to see which ones you like.

The `use_synth` function routine changes the sound of all notes until the next `use_synth` routine is used. It is possible to change the sound of notes as many times as you like within a song. We can even play a trick similar to the trick that we used when playing a random note. Try adding this line of code inside your loop:

```
use_synth choose([:pretty_bell, :dsaw, :hollow])
```

Again, feel free to add or remove any elements from this list. You should now hear random notes being played in random styles.

We can also change the length of time for which each note plays. There are three main parts to this:

- **attack**: This is the number of seconds taken to increase the volume to the maximum
- **sustain**: This is the number of seconds spent at maximum volume
- **release**: This is the number of seconds taken to decrease the volume to zero

The sum of all three of these times will give the total duration of our sound. The default `attack` and `sustain` values are zero; `release` is the most important value in most cases.

To use these features, add them at the end of the play command. You can use all three of them or none, or any number in between. Here's an example:

```
play 60, attack: 0.5, sustain: 0.1, release: 1
```

See how each option is separated with a comma and the option's name and its value are separated by a colon.

A real tune

So, we've had fun creating random tunes; now let's try making something a little more structured. Open a fresh workspace by clicking on one of the **Workspace** tabs underneath the Code Editor. This is where we'll create our new tune. You can switch back to your previous program any time by clicking on its tab.

In this section, we're going to build on the same piece of code, getting the sound we produce closer and closer to a tune you might know. How far will you get before you recognize the tune? A complete code listing is at the end of this chapter just in case you get stuck.

Here's the very first version of the code— you might recognize it already!

```
play :G
sleep 1
play :G
sleep 1
play :G
sleep 1
play :Eb
sleep 1
play :Bb
sleep 1
play :G
sleep 1
play :Eb
sleep 1
play :Bb
sleep 1
play :G
sleep 1
```

This code is simple but long. It's very easy to miss a line, and it's boring to write `sleep 1` so many times. Let's tidy it up a little by writing a function to do the work for us. This function will take a list of all the notes we want to play and go through them in sequence, playing each note and then sleeping. By doing this, it will be much easier to extend the song later.

We've seen similar functions before in Python. Here's how they look in Sonic Pi:

```
define :play_notes do |notes|
  notes.each do |note|
    play note
    sleep 1
  end
end
```

This function (and the short snippets we'll add soon) can replace all of the code we had previously. On the first line, we use `define` to define the function. We call it `:play_notes` (function names must begin with a colon), and we have an argument called `notes`. If we wanted multiple arguments, we would have separated them with commas between the two pipes. Finally, there is a `do` to show that this is the start of a code block.

On the second line, we begin our `loop`. We say that we want to look at each element in the list called `notes` and that inside the loop body, we will call the current element of the list `note`. Again, we have a `do` to show that we are starting the loop code block.

The next two lines form the body of the loop and should look familiar to you; we are playing the current note from the list and sleeping for one second.

The final two lines end the code blocks that we are in. We close the most recently opened block first; the first `end` closes the loop, and the second `end` closes the function definition.

Now, we need a list of notes to pass to the function. These are the same notes we've seen already, but held in a list instead of played individually:

```
line1 = [:G3, :G3, :G3, :Eb3, :Bb3, :G3, :Eb3, :Bb3, :G3]
```

Finally, we call the function using our list as the argument:

```
play_notes line1
```

If you run the code now, it should sound exactly the same as it did before. However, the program is now much shorter and easier to modify. For example, let's say that we decide that we don't want `sleep` for one second; we want `sleep` for some other length of time. Previously, we would have had to change the nine `sleep` commands, but now, there's only one `sleep` command to change. This makes our life easier, and also reduces the risk of forgetting to make all the necessary changes.

I think 0.8 is a good `sleep` period for this tune—make this change now. If you run this code, you'll notice that it sounds a little strange. Each note plays for one second, but the next note starts earlier than that, so there is an overlap. We can fix this by adding `release: 0.8` to the `play` command to reduce the length of each note (remember to separate it from the rest of the `play` command with a comma).

Now we have a similar problem to before, we're using the same number in multiple places and we want to make sure that they're always the same, just in case we make changes later. Let's store this number in a **variable** and use the variable name inside the loop. Before defining the function, add a line of code, like this:

```
pace = 0.8
```

Then, replace all the 0.8 numbers inside the function with `pace`.

Let's make one more change before we continue; we want to use a more appropriate sound than a simple beep. Add this line of code before calling the function:

```
use_synth :dsaw
```

Your program should now look something like this:

```
pace = 0.8

define :play_notes do |notes|
  notes.each do |note|
    play note, release: pace
    sleep pace
  end
end

line1 = [:G3, :G3, :G3, :Eb3, :Bb3, :G3, :Eb3, :Bb3, :G3]

use_synth :dsaw

play_notes line1
```

Adding rhythm

Do you recognize the tune yet? One of the most important missing elements is rhythm – all the notes are currently played in an identical way.

Let's add another list of numbers that represents how long each note should be played. We'll use 1 to represent a normal-length note, 2 to represent one that lasts twice as long, and 0.5 for a half-length note. The actual length of the note will be controlled by our pace variable. Add this code just above the `line1` list:

```
rhythm = [1, 1, 1, 0.75, 0.25, 1, 0.75, 0.25, 2]
```

Now, if we can modify our `:play_notes` function to play each note in `line1` for the amount of time shown in `rhythm`, our tune should sound much better. Here's a new version of the function, with the changes highlighted in bold:

```
define :play_notes do |notes, durations|
  together = notes.zip(durations)
  together.each do |note, duration|
    play note, release: pace * duration
    sleep pace * duration
  end
end
```

In the first line, we add an extra argument to the function, called `durations`. In the second line, we use the `zip` function to merge the two arguments into a single list, like this:

```
lista = [a1, a2, a3]
listb = [b1, b2, b3]
lista.zip(listb) = [ (a1, b1), (a2, b2), (a3, b3) ]
```

As you can see, `zip` pairs the first element of each list, the second element of each list, and so on. This allows us to access a note and its duration at the same time in the third line. Finally, inside the loop body, we multiply the pace by the duration of each particular note.

The last thing to change is the line where we run the `:play_notes` function. Now that the function takes two arguments, we need to provide it with two. Change the final line of your program so that it says this:

```
play_notes line1, rhythm
```

If you play the tune now, it should sound much more recognizable. What's more, we can easily add a second line to the tune now that we have everything set up. The rhythm will be the same, with only the notes being different. Add the following lines to the end of your program:

```
line2 = [:D4, :D4, :D4, :Eb4, :Bb3, :G3, :Eb3, :Bb3, :G3]
play_notes line2, rhythm
```

Your program should now look something like the following. I've done only a small amount of rearrangement to put similar pieces of information close to each other:

```
pace = 0.8

define :play_notes do |notes, durations|
  together = notes.zip(durations)
  together.each do |note, duration|
    play note, release: pace * duration
    sleep pace * duration
  end
end

rhythm = [1, 1, 1, 0.75, 0.25, 1, 0.75, 0.25, 2]
line1 = [:G3, :G3, :G3, :Eb3, :Bb3, :G3, :Eb3, :Bb3, :G3]
line2 = [:D4, :D4, :D4, :Eb4, :Bb3, :G3, :Eb3, :Bb3, :G3]

use_synth :dsaw

play_notes line1, rhythm
play_notes line2, rhythm
```

Bass line

We now have a nice tune, but what happens if we want to add a bass line to it, or any sort of sound that plays at the same time as the melody?

We could, of course, write our program so that every note played is either part of the melody or the bass line, but this will be very difficult to do, and there will be lots of effort spent switching between sound styles. Instead, we're going to use something called a **thread**. A thread allows one piece of code to run at the same time as another piece of code. We can have as many threads as we like in a program, to allow any number of pieces of code to run together.

To create a thread in Sonic Pi, use the `in_thread` code block:

```
in_thread do
  use_synth :dsaw

  play_notes line1, rhythm
  play_notes line2, rhythm
end
```

Notice that we don't have to put much of the program inside the thread. Function definitions and our lists can stay outside, in case any other threads want to access them. Our function calls go inside because these are the things that we want to happen at the same time as other parts of the music. Interestingly, we also put the `use_synth` syntax inside the thread—each thread can have its own synthesizer, so we do not need to change it every time a different thread plays a note.

Let's now switch to a fresh workspace so that we can work on the bass line without the melody getting in the way. Click on a new tab underneath the Code Editor.

The first thing we want to do is find a sound that can be our drumbeat. Some synthesizers can do quite well at this, but Sonic Pi has another feature that can do even better—**samples**. A sample is a piece of a sound recording (rather than a sound generated by the computer) and Sonic Pi has samples of several different instruments and lots of other things. To play a sample, type in `sample` and then press the spacebar. Again, you'll see a long list of available sounds. Double-click on one and click on **Run** to see how it sounds. Try this with as many samples as you like. You can even record your own samples by clicking on the **Rec** button at the top of the window.

For this tune, I think `:drum_tom_lo_hard` is a good sample to use, so use this one when you have finished experimenting. If you click on **Run**, you'll hear that this is a fairly normal drumbeat. We will use this beat to form our bass line.

However, the sound lasts quite a long time and we will want it to go a little quicker for our program. The `sample` function allows us to choose the `rate` at which a sound is played; the default is 1. A higher number means the sound is played more quickly, and a lower number means the sound is played slower. Try this out using the following line:

```
sample :drum_tom_lo_hard, rate: 2
```

Also feel free to try other rates and other samples. You'll notice that besides the length of the sound changing, the pitch changes as well. Unfortunately, this isn't what we want—the pitch of the original drum beat was good. Instead, we're going to return to the same attack, sustain, and release options as we saw with `use_synth`:

```
sample :drum_tom_lo_hard, attack: 0, sustain: 0, release: 0.2
```

We now have a drum beat at the normal pitch, but the sound doesn't last very long, and so it sounds punchier. We're going to be using this sound a lot, so it will be useful to put it inside a function that is dedicated to playing just this sound. This approach will give us a much shorter name for the sound, and so make our code easier to read. Here's the function:

```
define :drum do
  sample :drum_tom_lo_hard, attack: 0, sustain: 0, release: 0.2
end
```

See how this function doesn't have any arguments at all—there are no vertical bars on the first line. This means that this function always does exactly the same thing, which is what we want. To play the beat now, we just need to type `drum`.

We're now going to use a similar technique to the one we used with the melody to play our drum with a given rhythm. We'll use a list that tells us how long to wait between drumbeats and we'll use a function to take that list and play our drum sound. The function only takes the rhythm this time. It doesn't need any notes because every drumbeat is the same. Here's the additional code needed to do this:

```
pace = 0.8

drum_rhythm = [1,
               0.625,
               0.125, 0.125, 0.125, 0.5,
               0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.75]

define :play_beat do |beat|
  beat.each do |pause|
    drum
    sleep pace * pause
  end
end

play_beat drum_rhythm
```

Here, `drum_rhythm` is our list of times between each beat. There are lots of beats, so I've spread them over multiple lines and the beats are quite close together, so the numbers are small.

We also have the `:play_beat` function, which is very similar to the `:play_notes` function we used previously. Since the function uses `pace`—but `pace` was created in a different tab, and so it is not accessible from here—I have copied `pace` and placed it at the top.

Finally, we run our function with the list, and you can listen to the result by clicking on **Run**.

We're now ready to copy our bass line code to our program containing the melody. Copy all of the code in this tab, switch to the tab containing the melody, and paste the code at the bottom. Delete the extra `pace = 0.8` line—we only need the original one. If you click on **Run**, you should now hear the melody and bass line playing together!

But wait, it doesn't sound quite right! The bass line finishes much sooner than the melody. We need to put the bass line inside a loop to play it multiple times. Here, we have a choice. We can either play the bass line four times so that it lasts for exactly the same amount of time as the melody, or we can play the bass and melody forever. I'm going to use the first option in this example:

```
4.times do
  play_beat drum_rhythm
end
```

More fun

Our tune is now finished and hopefully you recognize it, so the rest of this section shows you some more fun features of Sonic Pi. All of these are optional. I don't think they improve this particular tune, but they will definitely be useful if you go on to create your own music.

So far, we've seen that Sonic Pi shares features such as loops, lists, functions, and randomness with Python (and almost every other programming language). One main shared feature that we haven't used yet in Sonic Pi is `if`. This controls whether something happens or not.

For this example, we're going to randomly choose whether each note is played out of the left or the right speaker.

Inside the `:play_notes` function, take a look at the `play` line. It has another option called `pan` to select which speaker the sound plays from. The default is 0, where both speakers play. The value -1 means only the left speaker, and 1 means only the right speaker. Any number in between is also possible. Here's some code that can go inside the loop to randomly choose a speaker:

```
if one_in(2)
  speaker = -1
else
  speaker = 1
end
```

The `one_in(2)` function behaves like a coin toss; it will give `true` one in every two times we use it (on average) and `false` the rest of the time. We can also mimic a six-sided die using `one_in(6)` and any other positive number works as well. The `if` block contains what happens if `one_in` gives us `true` and the `else` block tells what will happen if it gives `false`.

Now that we have this code, we just need to modify the `play` line to use this speaker value:

```
play note, release: pace * duration, pan: speaker
```

Code listing

This section gives a complete listing of the code used in this chapter. You can refer to it if your code is not working to see what changes need to be made. Some of the code blocks have been moved around within the program to keep similar parts of the program together. This can make things easier to read, but it is completely optional. The following is the complete code used in the chapter:

```
pace = 0.8

define :play_notes do |notes, durations|
  together = notes.zip(durations)
  together.each do |note, duration|
    if one_in(2)
      speaker = -1
    else
      speaker = 1
    end
    play note, release: pace * duration, pan: speaker
  end
end
```

```
        sleep pace * duration
      end
    end

    define :drum do
      sample :drum_tom_lo_hard, attack: 0, sustain: 0, release: 0.2
    end

    define :play_beat do |beat|
      beat.each do |pause|
        drum
        sleep pace * pause
      end
    end

    rhythm = [1, 1, 1, 0.75, 0.25, 1, 0.75, 0.25, 2]
    line1 = [:G3, :G3, :G3, :Eb3, :Bb3, :G3, :Eb3, :Bb3, :G3]
    line2 = [:D4, :D4, :D4, :Eb4, :Bb3, :G3, :Eb3, :Bb3, :G3]

    drum_rhythm = [1,
                   0.625,
                   0.125, 0.125, 0.125, 0.5,
                   0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.75]

    in_thread do
      use_synth :dsaw

      play_notes line1, rhythm
      play_notes line2, rhythm
    end

    4.times do
      play_beat drum_rhythm
    end
  end
```

Summary

In this chapter, we used Sonic Pi to create our own music. We went from random sounds to a full tune with a drumbeat and saw how the programming ideas you learned in previous chapters can be used naturally to describe music.

Throughout this book, you have learned about the Raspberry Pi and what it can be used for. You also learned some core programming concepts and saw how they apply to Scratch, Python, and Sonic Pi. They apply to many other programming languages too. We saw how programming can be a creative skill and can be used to create games or build useful tools. Above all, I hope you've found programming fun. It's a really valuable skill to learn and can provide unlimited entertainment.

If you've enjoyed this book and would like to continue your Raspberry Pi exploration, here are a few related books from Packt Publishing that you might find interesting:

- *Scratch 1.4: Beginner's Guide*
- *Raspberry Pi Cookbook for Python Programmers*
- *Instant Minecraft: Pi Edition Coding How-to*
- *Raspberry Pi for Secret Agents*

Index

A

Adafruit

URL 68

address

obtaining, for map 88, 89

Angry Birds™ game

about 37

finishing 47

level, creating 41

physics, adding 46

scoring 50

animation

creating, with Scratch 21

B

bind 92

C

Canvas widget 90

character, Angry Birds™ game

initializing 42

launching 44

moving, with keyboard 42

Checkbox 101

code blocks, Scratch

control 24

looks 24

motion 24

operators 24

pen 25

sensing 24

sound 24

URL 24

variables 25

Code Editor 56

conditionals 61

D

dictionary 59

E

extensions, Angry Birds™ game

adding 51

F

Frame 102

function

about 62

calling 54

defining 54, 63, 64

G

game, coding

about 72

completing 79

controller, using 74, 75

implementing 77, 78

random behavior 73

time limit, adding 75, 76

game controller

buttons, adding 69-71

connecting, to Raspberry Pi 71, 72

controller base 69

creating 68

requisites 67, 68

General-Purpose Input/Output (GPIO) 71

Google Maps

about 87
references 87

Graphical User Interface (GUI) 83

grid layout 101

H

Hello world! program

about 84
writing 84-86

I

If-then-else method 34

image

downloading, for map 89

interactive animation

about 29, 30
If-then-else method, using 34, 35
movements, adding 32
sprite count 33, 34
variables 31

K

key 59

keyboard version 80

key-value pair 60

L

LabelFrame 102

labels

adding 94
basic labels 94
pop-up windows 95-98

layout

about 84, 101
grid layout 101
pack layout 101

Listbox 102

lists 56

loop 60

M

map

address, generating for 88, 89

image, downloading for 89

image, using for 90, 92

obtaining 87

markers

adding 92

material requisites, Raspberry Pi

about 2
inputs 5
network 7
power supply 3
storage 4
videos 6

Menu 102

Menubutton 103

Message 103

mouse clicks

detecting 92
reacting to 93, 94

N

NOOBS

URL 9

O

OpenELEC 18

OptionMenu 103

overscan settings 19

P

package system 17

pack layout 101

phrases

creating 58
generating, programs used 55
lists 56
randomness, adding 57, 58

physics, Angry Birds™ game

adding 46
bounce, adding 46, 47
gravity, adding 46

Pi breakout board 68

Python

about 53, 54
function, calling 54
modules, URL 57

programming 55
references 55, 87

R

Radiobutton 104

Raspberry Pi

about 1
common issues, troubleshooting 19
forums, URL 19
game controller, connecting to 71, 72
ls command 16
material requisites 2
OS, installing 9
software, installing 17
software, updating 17
starting up 11-14
URL, for checking device compatibility 3
using 14

Raspberry Pi Swag

URL 4

RaspBMC 18

real tune

about 112-114
bass line 116-119
fun features 119
rhythm, adding 115, 116

RPi VerifiedPeripherals

URL 6

S

samples 117

Scale 104

Scratch

about 21-23
animation, alternative way 28, 29
code blocks 24, 25
elements 22, 23
Hello world! program, creating 23, 24
rotating cat 25, 26
scene, setting 26, 27
URL 21
used, for creating animation 21

SD card

preparing 8-10

SD Formatter

URL 9

Shell 56

Sonic Pi

about 107, 108
sound, creating 108, 109
URL 108

sounds 111

Spinbox 104

synthesizer 111

T

Tab key 16

thread 116

Tkinter

about 84
URL 100

Tk toolkit 84

tune

creating 109, 110

U

Uniform Resource Locator (URL) 89

V

value 59

variable 114

W

widgets

about 84, 101
Checkbutton 101
Frame 102
LabelFrame 102
Listbox 102
Menu 102
Menubutton 103
Message 103
OptionMenu 103
Radiobutton 104
Scale 104
Spinbox 104



Thank you for buying **Raspberry Pi Projects for Kids** *Second Edition*

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

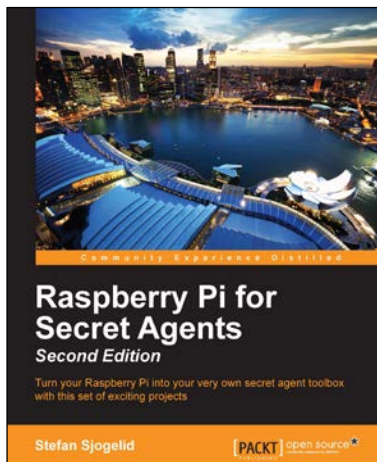
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Raspberry Pi for Secret Agents *Second Edition*

ISBN: 978-1-78439-790-6

Paperback: 206 pages

Turn your Raspberry Pi into your very own secret agent toolbox with this set of exciting projects

1. Turn your Raspberry Pi into a multipurpose secret agent gadget for audio/video surveillance, Wi-Fi exploration, or playing pranks on your friends.
2. Detect an intruder on camera and set off an alarm and also find out what the other computers on your network are up to.
3. Full of fun, practical examples and easy-to-follow recipes, guaranteeing maximum mischief for all skill levels.



Raspberry Pi Robotic Projects

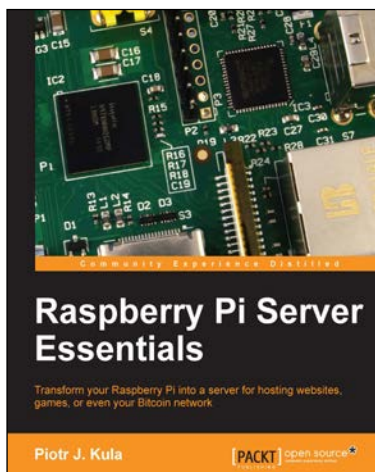
ISBN: 978-1-84969-432-2

Paperback: 278 pages

Create amazing robotic projects on a shoestring budget

1. Make your projects talk and understand speech with Raspberry Pi.
2. Use standard webcam to make your projects see and enhance vision capabilities.
3. Full of simple, easy-to-understand instructions to bring your Raspberry Pi online for developing robotics projects.

Please check www.PacktPub.com for information on our titles

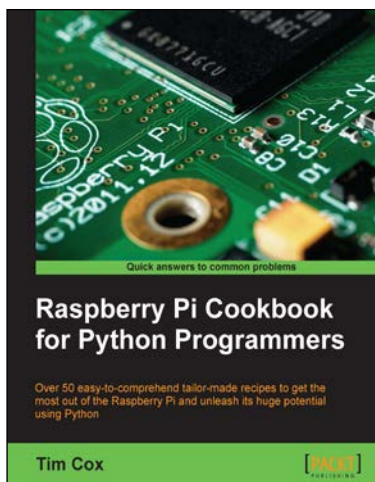


Raspberry Pi Server Essentials

ISBN: 978-1-78328-469-6 Paperback: 116 pages

Transform your Raspberry Pi into a server for hosting websites, games, or even your Bitcoin network

1. Unlock the various possibilities of using Raspberry Pi as a server.
2. Configure a media center for your home or sharing with friends.
3. Connect to the Bitcoin network and manage your wallet.



Raspberry Pi Cookbook for Python Programmers

ISBN: 978-1-84969-662-3 Paperback: 402 pages

Over 50 easy-to-comprehend tailor-made recipes to get the most out of the Raspberry Pi and unleash its huge potential using Python

1. Install your first operating system, share files over the network, and run programs remotely.
2. Unleash the hidden potential of the Raspberry Pi's powerful Video Core IV graphics processor with your own hardware accelerated 3D graphics.
3. Discover how to create your own electronic circuits to interact with the Raspberry Pi.

Please check www.PacktPub.com for information on our titles